Estruturas de Dados

Domine as práticas essenciais em C, Java, C#, Python e JavaScript

Casa do Código | alura

THIAGO LEITE E CARVALHO

Sumário

- ISBN
- Dedicatória
- Sobre o autor
- Prefácio
- Sobre o livro
- Parte 1: Dentro da Matrix
  - 1 Introdução
  - 2 Conceitos básicos
  - 3 Vetor e Matriz
  - 4 Pilha
  - 5 Fila
  - 6 Lista
  - 7 Vetor/Matriz vs. Pilha vs. Fila vs. Lista
  - 8 Árvore
  - 9 Grafo
  - 10 Outras estruturas de grande relevância
- Parte 2: O mundo real
  - 11 Estruturas de Dados em Java
  - 12 Estruturas de Dados em C#
  - 13 Estruturas de Dados em Python
  - 14 Estruturas de Dados em JavaScript
- Parte 3: Conclusão e apêndices
  - 15 Chegamos ao fim
  - 16 Referências bibliográficas
  - 17 Apêndice I: Ponteiro em C
  - 18 Apêndice II: Struct em C
  - 19 Apêndice III: Funções nativas
  - 20 Apêndice IV: Recursividade

ISBN

Impresso: 978-85-5519-338-5

Digital: 978-85-5519-339-2

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Carvalho, Thiago Leite e
Estruturas de dados : domine as práticas essenciais em C, Java, C#, Python e JavaScript / Thiago Leite e Carvalho. -- São Paulo : AOVS Sistemas de Informática, 2023.

Bibliografia.
ISBN 978-85-5519-338-5

1. Algoritmos de computadores 2. Ciência da computação 3. Dados - Estruturas (Ciência da computação) 4. Inteligência artificial 5. Linguagens de programação (Computadores) I. Título.

23-157417 CDD-005.73

Índices para catálogo sistemático:

1. Dados : Estruturas : Processamento de dados 005.73

Eliane de Freitas Leite - Bibliotecária - CRB 8/8415

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Dedicatória

Dedico mais este livro aos que sempre estão ao meu lado: João, Rogéria, Lucélia, Sarah, Isadora e Lorena. Além destes, a todos os meus colegas de profissão e faculdade que me ajudaram a me tornar o profissional de sucesso que sou hoje.

Agradecimento especial a Maikol Rodrigues, meu professor (de Estrutura de Dados!) nos tempos da faculdade e, hoje, colega de trabalho. Muitas foram as conversas e dicas durante a escrita deste livro. Um agradecimento especial ao Hugo Benício, também colega de trabalho e que sempre foi solícito quando surgiam dificuldades com as linguagens C e Python. Obrigado pelo apoio e o incentivo de vocês na escrita de mais um livro. Grande abraço!

Sobre o autor

Olá, pessoal! Meu nome é Thiago Leite e Carvalho. Adoro desenvolvimento e trabalho com isso desde 2003. Desde os estágios no tempo de faculdade até hoje, já trabalhei em empresas de vários ramos e tipos: software house, empresas públicas, no ramo da saúde, indústrias, entre outros. Também já prestei algumas consultorias focadas no desenvolvimento. Sou graduado e mestre em Engenharia de Software pela Universidade de Fortaleza. Já embarquei no mundo acadêmico e por dez anos fui professor de algumas faculdades, ministrando cadeiras de Programação Orientada a Objetos I e II; Engenharia de Software; Projeto e Arquitetura de Software; Linguagens Formais e Autômatos; Compiladores e Estruturas de Dados. Adoro lecionar e parto do princípio de que a melhor forma de aprender é ensinar.

Profissionalmente, sou programador Java e possuo conhecimento nos frameworks deste universo: Spring, Hibernate, JSF, Struts etc. Possuo três certificações em Java. Também já fui desenvolvedor C# e possuo conhecimentos em Python e Angular. Resumindo, sou um entusiasta do desenvolvimento de software. Atualmente, sou funcionário público, trabalhando no Serpro, empresa de tecnologia do Governo Federal do Brasil. Dedico-me a escrever livros, artigos para o LinkedIn e a produzir cursos para a Udemy. Além disso, também ministro cursos e palestras para instituições. Caso queira saber um pouco mais sobre mim, acesse o meu perfil no LinkedIn: https://www.linkedin.com/in/thiago-leite-e-carvalho-1b337b127/.

Prefácio

As estruturas de dados são fundamentais para qualquer pessoa desenvolvedora, pois são a base para a construção de algoritmos eficientes e sistemas computacionais robustos. Elas permitem a programadores e programadoras armazenarem e manipularem dados de forma organizada e otimizada, o que resulta em programas mais rápidos e eficientes. Além disso, as estruturas de dados são essenciais para a solução de problemas complexos em diversas áreas, como na inteligência artificial, onde pode ser usada em *Machine Learning*; no processamento de imagens, para a detecção de padrões; na análise de dados, para avaliações quantitativas e analíticas; na Engenharia de Software, para a construção de soluções que visam à automatização de atividades e processos, entre diversas outras. Ou seja, qualquer que seja o curso (Engenharia da Computação, Análise e Desenvolvimento de Sistemas ou Ciência da Computação), as Estruturas de Dados são uma teoria essencial para a formação de profissionais. Portanto, um entendimento eficaz das estruturas de dados é crucial para que os profissionais da área de computação desenvolvam habilidades sólidas e estejam preparados para enfrentar os desafios cada vez mais complexos na área da Tecnologia da Informação.

Conheço o Thiago desde 2001 e venho acompanhando o seu caminhar desde a época da universidade, quando ele foi meu aluno nas disciplinas de Estruturas de Dados e Pesquisa Operacional. Atualmente somos colegas de trabalho: em 2012, comecei a trabalhar na mesma empresa que ele. Thiago sempre se destacou no desenvolvimento na linguagem C e em Java com Orientação a Objetos e, por isso, já foi convidado a ministrar em empresas e universidades diversos cursos, como Java, Hibernate, Spring, entre outros. Também já trabalhou com a plataforma .Net por 4 anos.

Com o início da carreira de professor universitário, juntamente com os anos de experiência em desenvolvimento, ele percebeu a dificuldade de seus alunos e alunas – e até mesmo de profissionais

com experiência em programação – de realmente compreenderem e aplicarem os conceitos práticos de Estruturas de Dados. Essas pessoas cometiam falhas na escolha das estruturas de dados mais adequadas e eficientes para a resolução de problemas específicos. Ele também percebeu que existia uma lacuna de textos didáticos, focados em explicar detalhadamente as estruturas de dados clássicas e também em abordá-las nas principais linguagens que o mercado utiliza. Por tudo isso, ele resolveu escrever este livro.

Thiago teve o cuidado de selecionar as principais e mais utilizadas estruturas de dados e apresentar, de forma clara e objetiva, os principais conceitos de cada uma delas. Além de vários exemplos (códigos) explicados de forma minuciosa, são apresentadas também implementações em várias linguagens de programação. A linguagem C é a mais utilizada no decorrer do livro, mas também são exploradas as linguagens Java, C#, Python e JavaScript. A organização dos conceitos de forma fluida e gradativa facilita o aprendizado e desmistifica a complexidade atribuída a esse assunto.

Um grande diferencial deste livro é que ele aborda as estruturas de dados (ED) de forma simples, concisa e com exemplos do nosso dia a dia, mostrando que elas são mais naturais do que imaginamos. O livro inicia com conceitos básicos e inerentes a Estruturas de Dados e Computação, passando pelas EDs mais básicas e seguindo até as mais avançadas. Nessa trilha, sempre são fornecidos textos e códigos comentados com o intuito de facilitar a absorção do conteúdo. Para finalizar, vários exercícios são apresentados para complementar o aprendizado e todos eles estão resolvidos e disponíveis para leitores e leitoras.

Tenho certeza de que, ao terminar a leitura deste livro, você será uma pessoa desenvolvedora diferenciada e mais preparada para usar as estruturas de dados da melhor forma possível. Embarque com Thiago nesta desafiadora e engrandecedora caminhada de como organizar os dados e ter melhor desempenho nos seus algoritmos. Boa leitura!

— ***Maikol Magalhães Rodrigues***

Sobre o livro

É muito comum iniciantes no mundo da computação terem dificuldades no entendimento dos conceitos sobre *estruturas de dados*. Geralmente, isso é decorrente de um maior nível de abstração exigido para compreender o funcionamento de tais estruturas, nível que nem sempre se está preparado para assimilar e compreender. Outro fator recorrente é que mesmo pessoas com certo "nível avançado" na computação também têm dificuldade de utilizar as estruturas de dados existentes de forma efetiva e satisfatória, sendo isso um efeito colateral também relativo à dificuldade de assimilação e compreensão.

Somada a essas "questões pessoais", ainda temos a dificuldade de encontrar livros dedicados em apresentar e explicar tais estruturas de forma *minuciosa* e *focada*, assim como todos os conceitos que também fazem parte do universo das estruturas de dados. Tudo isso termina por dificultar a aplicação adequada das estruturas de dados no desenvolvimento de software. Por fim, se for acrescido a estes problemas o fato de que cada linguagem tem suas próprias implementações de tais estruturas, temos um grande problema que pode ocasionar graves erros no processo de codificação.

É com base nesses fatos que este livro tem como principal intuito expor, da forma mais didática e simplificada possível, todos os conceitos pertences às estruturas de dados. Para atingir este objetivo, o livro é dividido em duas partes:

- **Parte 1 - Dentro da Matrix**: Nesta primeira parte, as "entranhas" das estruturas de dados serão expostas para conseguirmos compreender seus funcionamentos e aplicabilidades.

- **Parte 2 - O mundo real**: Após os fundamentos terem sido expostos, é apresentado como as principais linguagens do mercado (Java, C#, Python e JavaScript) implementam tais estruturas. Nessa parte, toda a API destas linguagens são exploradas para facilitar o uso das estruturas de dados.

É válido ressaltar que, para facilitar o alcance destes objetivos, este livro foca exclusivamente nas estruturas de dados. Ou seja, um conceito recorrentemente exposto de forma conjunta como *busca e ordenação* não estará presente neste livro. Todavia, ele pode ser encontrado em literaturas específicas de *algoritmos* e *complexidade de algoritmos*. Outra observação relevante ao modo como este livro é organizado é o fato de que, na Parte 1, as explicações sobre as teorias das estruturas de dados são mais detalhadas; por outro lado, na Parte 2, que foca nas linguagens, apenas focamos na definição e no uso de tais estruturas. Assim, temos a possibilidade de explorar algumas nuanças de cada linguagem para tais estruturas.

Por fim, os códigos de exemplos e exercícios da Parte 1 deste livro são feitos na linguagem C usando a ferramenta Eclipse IDE for C/C++ Developers. Tal escolha vem do fato de que, com esta linguagem, conseguimos apresentar de forma mais minuciosa como a implementação das estruturas ocorrem em suas entranhas.

Este livro não é um "curso de C", então é de se esperar que o leitor já tenha um conhecimento prévio desta linguagem, pois muitos dos detalhes do uso dela não são explicados. Todavia, alguns dos conceitos presentes em C e que são relevantes para o livro são explicados de forma mais detalhada em alguns apêndices. Caso necessário, leia-os.

Também é válido frisar que, quando as estruturas de dados nas linguagens Java, C#, Python e JavaScript forem expostas, elas serão integralmente apresentadas nestas linguagens, através de códigos nativos de tais linguagens ou implementadas "à mão", caso necessário. Os códigos nestas linguagens foram criados de forma a facilitar ao máximo o entendimento deles. Talvez, com o passar do

tempo, note-se que eles podem ser melhorados — e sugiro isso como exercício. Todos os códigos (completos e relevantes) aqui expostos estarão no meu perfil do GitHub (em https://github.com/thiagoleiteecarvalho), nos repositórios que iniciam com "*ed*".

Espero que a leitura seja agradável e enriquecedora. Com a ajuda da Casa do Código, trabalhei de forma árdua para explicar, da forma mais amigável e instigante, os conceitos das estruturas de dados. Este livro é a realização de um projeto que visa fornecer bases sólidas para o uso e entendimento delas. Agradeço a escolha deste livro.

"*Muito a aprender você ainda tem*." — Mestre Yoda

Parte 1: Dentro da Matrix

Nesta primeira parte do livro, serão apresentados todos os conceitos inerentes às estrututras de dados, assim como os seus tipos e subtipos, além de conceitos relevantes, mas que não fazem parte diretamente da teoria das Estruturas de Dados. Assim, espera-se propiciar a quem lê a capacidade de entender como elas funcionam "internamente" para identificar o momento certo de usar cada uma delas.

Capítulo 1
Introdução

— *Poxa, a fila no hospital vai estar grande.*

— *Meu filho! Não pegue o prato mais abaixo da pilha, você pode quebrá-los!*

— *Onde está a lista com os passos para montar a estante?*

Inicialmente, podemos pensar que essas frases estão fora de contexto, mas, na verdade, não. Embora elas representem fatos corriqueiros do nosso dia a dia, elas têm — disfarçadamente — um pouco de "computação" nelas, mais precisamente de *estruturas de dados*.

A *fila* do hospital, a *pilha* de pratos e a *lista* de passos são materializações no mundo real do que também existe dentro dos computadores. Essas são apenas três de várias outras estruturas de dados que existem na computação e que uma vez ou outra vemos no nosso dia a dia.

Mas o que são **estruturas de dados**? De forma sucinta, podemos dizer que são:

Formas como os dados podem ser armazenados e organizados para posterior uso.

Ou seja, de nada adianta conseguirmos armazenar os dados, se eles estiverem de forma desorganizada. Isso dificultaria muito a manipulação deles. Imagine que, ao ir ao hospital, não existisse uma fila. Que todos "lutassem" para serem atendidos na *ordem* que quisessem. Seria um caos! Dentro dos computadores não é diferente, e a *ordem* anteriormente citada é algo que contribui muito no modo como eles processam os dados. A finalidade das estruturas de dados é propiciar que os dados sejam manipulados de forma organizada para que se possa usá-los de forma a se obter o máximo de benefício.

Com essa definição inicial e simples do que são estruturas de dados, você já deve imaginar como elas impactam drasticamente a nossa vida. A presença da computação é constante em nosso cotidiano e ela usa de forma abundante as estruturas de dados para a realização de processos e, consequentemente, para automatizações que nos possibilitam executar com celeridade e segurança muitas de nossas atividades. E para nós, como pessoas desenvolvedoras de software, quais os impactos? Eles são maiores ainda!

Somos responsáveis por criar os softwares que automatizarão e agilizarão a execução de atividades antes demoradas e maçantes. Entender com precisão como funcionam tais estruturas e saber usá-las da forma adequada é primordial. Embora inicialmente sejam assuntos complexos, com o passar do tempo chegaremos à conclusão de que, na verdade, são praticamente inerentes ao nosso dia a dia, seja como consumidor ou criador dessas estruturas. Ao encará-las dessa forma, seu entendimento se tornará mais natural e acessível, o que possibilitará uma utilização eficaz e eficiente, que resultará em softwares de alta qualidade. No fim, tudo será uma questão de amadurecimento do raciocínio lógico, que virá com tempo e prática.

Por fim, embora se possa pensar que, ao usarmos linguagens orientadas a objetos ou funcionais, podemos relevar o entendimento das "entranhas" das estruturas de dados, isso é um equívoco. Ainda que essas linguagens abstraiam muito a maneira como tais estruturas são criadas e isso torne seus usos mais fáceis, na verdade, entendê-las de modo claro ajuda a usá-las de forma precisa, além de ajudar no amadurecimento do raciocínio lógico. O resultado disso é o que foi já foi dito: softwares de alta qualidade.

Antes de mergulharmos nessas estruturas, é importante entendermos alguns conceitos-chaves que são as bases em que elas se alicerçam, como *dado*, *informação*, *memória*, entre outros. Que comecemos nossa caminhada!

Capítulo 2
Conceitos básicos

Neste capítulo, serão apresentados conceitos essenciais para o entendimento e funcionamento das estruturas de dados, tais como: dado, informação e tipo de dado. Veremos também conceitos que auxiliam no entendimento e no uso dessas estruturas, como: memória, tipos de estruturas e ponteiros.

2.1 O que é dado?

Um *dado* pode ser definido como *um valor bruto e sem significado*. `10`, `"b"`, `2.6`, `true` são exemplos de dados. Nota-se que estes, utilizados em sua forma bruta, não possuem serventia por serem demasiadamente abstratos e atômicos.

Dados são a base da computação, mais precisamente a manipulação deles. Tudo que um computador faz é manipular dados para nos auxiliar na tomada de decisões. Podemos dizer que dados podem ser originados a partir da interação entre usuários humanos com computadores (softwares) ou da interação direta entre computadores (softwares), seja apenas através de trocas ou da criação de novos dados a partir da manipulação de dados preexistentes.

2.2 O que é estrutura?

Uma *estrutura* pode ser definida como *a forma como um conjunto de dados pode ser armazenado e manipulado*. A depender do *tipo de dado* — conceito que será abordado mais adiante —, cada

estrutura é armazenada e manipulada de uma maneira específica. Assim, podemos dizer que cada estrutura define uma álgebra, um conjunto de operações/manipulações que são permitidas sobre esta estrutura.

2.3 O que são estruturas de dados?

Após definirmos o que é *dado* e o que é *estrutura*, uma pergunta pode surgir: o que são, então, as *estruturas de dados*? Podemos dizer que são

> *Um conjunto de teorias e práticas responsáveis por definir a forma como os dados podem ser armazenados, representados e consequentemente manipulados.*

É nessa área de estudo que são apresentadas as formas como os computadores e, consequentemente, os softwares podem definir e manipular os dados.

2.4 O que é informação?

Um conceito também relevante, mas não necessariamente pertencente à disciplina de Estruturas de Dados, é a definição de *informação*. Pode-se dizer que ela surge quando um *dado* ganha um significado, uma semântica a partir ou para um determinado contexto. Por exemplo, já citamos os seguintes exemplos de dados: `10`, `"b"`, `2.6`, `true`. Algumas informações que poderiam ser obtidas ou inferidas através deles seriam:

- O carro tem 10 anos de uso.
- A letra "b" é a resposta correta para a 3ª questão.

- A taxa de juros é de 2.6 ao mês.
- O seguro está habilitado: true.

2.5 O que é TD e TAD?

Tipo de dado (TD) pode ser definido como as categorias nas quais os dados podem se enquadrar. Os TDs com que os computadores trabalham podem ser divididos em:

- Numéricos
- Lógicos
- Literais

Algumas dessas categorias podem possuir subcategorias que visam aumentar o poder de representatividades. Por exemplo, numéricos podem ser *inteiros* ou *reais*. Literais podem ser *textos* ou *caracteres*. Lógicos, entretanto, apenas podem possuir dois valores: `true` e `false`. Essas três categorias são responsáveis por representar os **tipos primitivos**, que são os TDs mais básicos que as linguagens de programação podem utilizar.

É válido ressaltar que os TDs são fortemente ligados às linguagens de programação, e elas podem disponibilizar maiores subdivisões ou TDs próprios. Como exemplo de subdivisões, podemos citar Java, que disponibiliza para *inteiros* os tipos `byte`, `short`, `int` e `long`. A diferença entre eles está na capacidade de armazenamento. Como exemplo de TD próprio, podemos citar JS, que disponibiliza o tipo `symbol`, o qual representa um valor literal e imutável.

Ainda em relação aos *tipos de dados*, podemos verificar que algumas linguagens possuem uma "tipagem forte" ou "tipagem fraca". Isso diz respeito ao momento em que devemos informar o *tipo de dado* que a variável deve ser. Devido a isso variar muito de acordo com a linguagem, não exploraremos esse assunto aqui, mas aconselho muita atenção a isso quando for trabalhar com sua

linguagem favorita. Por fim, a partir da manipulação desses *tipos de dados*, podem ser criados TADs.

O *tipo abstrato de dado* (TAD) é um *tipo de dado* construído a partir de tipos primitivos (TDs) e tem como finalidade representar *estruturas* do mundo concreto (real) para o mundo abstrato (computacional). Assim, podemos afirmar que TADs — ao contrário dos TDs — não são ligados às linguagens de programação, embora possam ser representados por elas. Como dito, TADs são *estruturas* (conceitos/entidades) que pertencem a um contexto que deve ser modelado para poder ser manipulado pelo computador.

O termo "abstrato" é utilizado no sentido de ausência de detalhes, evitando uma descrição minuciosa do mundo concreto. Entretanto, é válido ressaltar que, mesmo com essa "ausência de detalhes", os TADs são capazes de alcançar os objetivos almejados no mundo computacional, pois essa simplificação não diminui a capacidade de representação, apenas torna o processo de representação mais fácil de entender pelo homem e pelos computadores. Essa simplificação é vital para tornar o mundo real "computável".

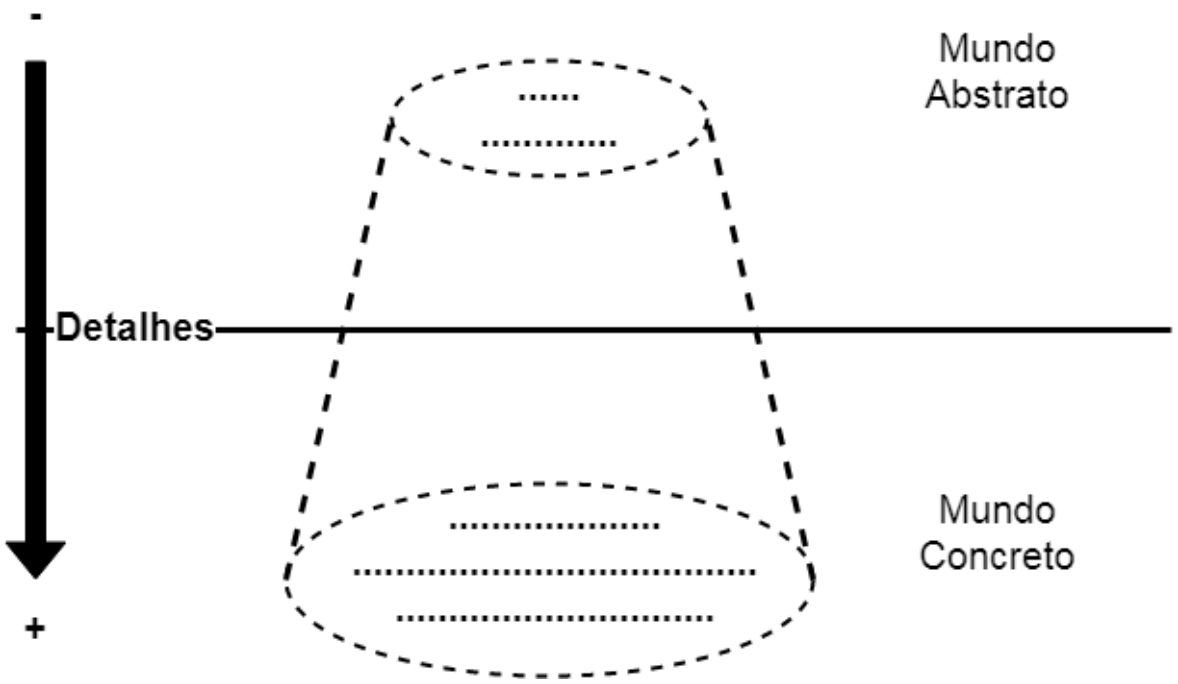


Figura 2.1: Modelo conceitual de um TAD.

Assim como os TDs, os TADs possuem suas categorias:

- Lineares
- Hierárquicos
- Mapas
- Conjuntos

Algumas dessas categorias também podem possuir subcategorias que visam aumentar o poder de representatividades. E é justamente neste ponto que começamos a adentrar no universo das estruturas de dados que este livro pretende explorar, as quais serão expostas mais adiante.

Para finalizar esta parte de conceituação de TD e TAD, podemos dizer que TAD pode ser considerado um TD, no caso um tipo *Complexo*, que pode ser encarado como uma evolução dos *tipos primitivos*, pois é constituído a partir destes e possui um poder representacional maior.

Álgebra sobre TD e TAD

Tendo TD e TAD sido definidos, é importante destacar que eles possuem uma álgebra definida para cada uma de suas categorias. A seguir, exemplificaremos as álgebras a partir dos *tipos primitivos*.

- Numéricos
  - Operações: `+`, `-`, `*`, `/` ;
  - Comparações: `=`, `>`, `<`, `≥`, `≤`, `≠`.
- Lógicos
  - Operações: conjunção (`and`, `∧`), disjunção (`or`, `∨`), disjunção exclusiva (`xor`, `⊻`), negação (`!`, `-`, `¬`);
  - Comparações: `=` e `≠`.
- Literais:
  - Operações: não se aplica;
  - Comparações: `=`, `>`, `<`, `≥`, `≤`, `≠`.

As álgebras dos TADs serão abordadas à medida que eles forem sendo apresentados no decorrer do livro (e há ainda outras que não pertencem ao escopo do livro, pois fazem parte de outra disciplina no universo da programação).

2.6 O que é um ponteiro?

É uma forma de acessar dados (TD ou TAD) através do local na memória do computador onde eles se encontram. Ponteiros possibilitam acesso ao valor do dado e também ao seu endereço de memória.

Inicialmente pode-se pensar que esse tipo de manipulação para dados é desnecessário, mas na verdade não é. Ponteiros são a base para a criação das estruturas de dados que serão abordadas neste livro. Com o uso deles, é possível alocar e desalocar memória dinamicamente.

Além disso, sempre estamos usando ponteiros, seja de forma direta ou indireta. Linguagens mais modernas como Java, C# e Python não possibilitam o uso direto de ponteiros; entretanto, toda vez que utilizamos uma variável que é do tipo de algum objeto (seja criado pelo programador ou disponibilizado pela própria linguagem), estamos usando um ponteiro indiretamente. Ou seja, por "debaixo dos panos", tais linguagens usam ponteiros para manipular objetos. Elas só fornecem um nível de abstração maior, que nos poupa de manusear diretamente ponteiros, eliminando possíveis erros decorrentes de acessos indevidos a determinados locais na memória.

Ao contrário dessas linguagens, C possibilita o manuseio desse tipo de acesso a dados. Inicialmente pode-se pensar que isso é ruim devido ao problema de acesso indevido, mas veremos que, a depender da necessidade, ponteiros podem ser a melhor forma de se manipular os dados. No contexto deste livro, isso ajuda a compreender as "entranhas" da ED e justamente por isso C foi a linguagem escolhida como base para os códigos deste livro.

> Para uma rápida revisão, aconselho a leitura do *Apêndice I: Ponteiros em C*, pois utilizaremos muito este conceito no decorrer do livro.

2.7 Memória

Para que todos os conceitos anteriormente citados possam ser utilizados pelo computador e, consequentemente, pelos softwares, eles devem ser disponibilizados através de algum mecanismo, que, neste caso, é a *memória*. Pode-se dizer que ela é "*um componente capaz de armazenar dados e programas (softwares)*". É nesse componente que os softwares são carregados para execução, assim como os dados que são manipulados pelos softwares.

Entretanto, para chegarmos ao ponto que nos interessa — *heap* e *stack* — dentro de todos os conceitos envolvidos no conceito de *memória*, vamos primeiro revisitar dois conceitos que são explorados na disciplina de Arquitetura de Computadores: Memória Principal (MP) e Memória Secundária (MS).

A MP é um tipo de memória que possibilita acesso rápido e que tem curto tempo de vida. Dados armazenados nesse tipo de memória são voláteis. Ao contrário dela, a MS possui um acesso mais lento e tem tempo de vida longo. Dados armazenados nesse tipo de memória são duráveis. Como principais exemplos dessas memórias temos a RAM e a ROM, respectivamente. Pela característica de ter acesso rápido e curto tempo de vida, a RAM é onde — preferencialmente — o *heap* e a *stack* são alocados, pois programas entram e saem de execução constantemente. Esse processo de alocação da memória para ser usada pelo *heap* ou *stack* pode ser feito de forma estática ou dinâmica, a depender da necessidade.

Na alocação estática, a memória de que um *tipo de dado* ou programa possa vir a necessitar é alocada toda de uma vez e de forma sequencial, sem considerar que toda ela não seria realmente necessária na execução do programa. A alocação dinâmica, por sua vez, aloca a memória sob demanda e de forma não sequencial. Assim, os espaços de memória podem ser alocados, liberados ou realocados para diferentes propósitos durante a execução do programa. O alocador de memória do Sistema Operacional (SO) aloca blocos de memória que estão livres e que são gerenciados por ele.

Heap

O *heap* é o principal espaço de memória utilizado pelos computadores para executar programas. É nesse local que linguagens estruturadas armazenam as variáveis de escopo global e que linguagens orientadas a objetos armazenam os objetos criados através do operador `new`. É também nesse local que o programa é carregado para poder ser executado, assim como todo o espaço

inicial de memória que precisar ser alocado, inclusive *stacks*. Esse espaço de memória utiliza constantemente a alocação dinâmica sob demanda para possibilitar a execução de programas. A figura a seguir apresenta uma imagem conceitual do funcionamento do *heap*.

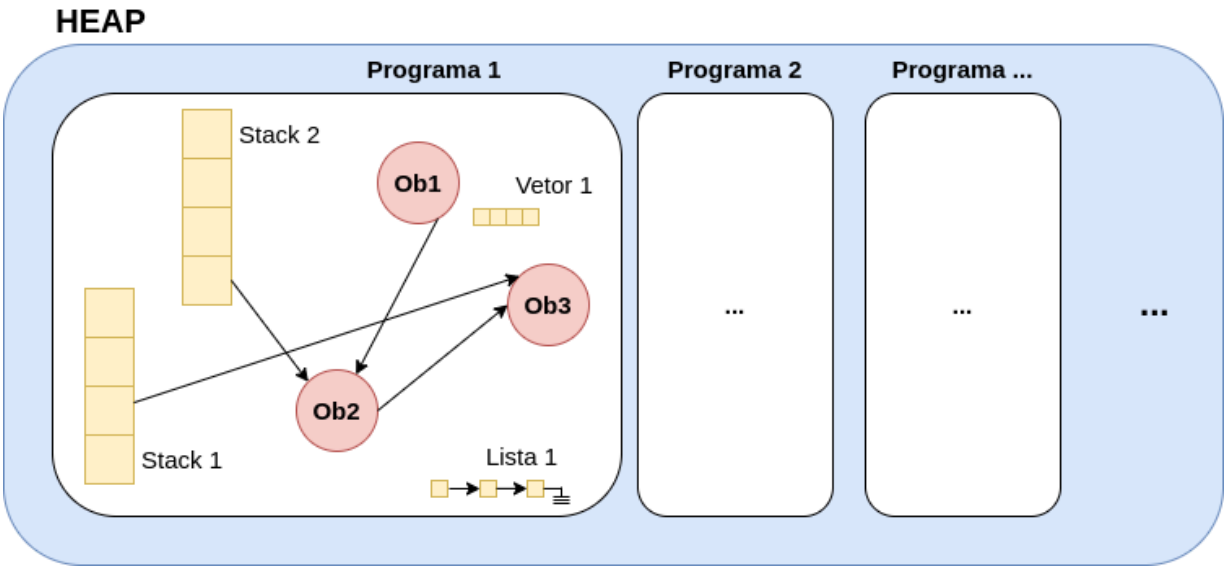


Figura 2.2: Modelo conceitual do funcionamento do heap de memória.

Stack

A *stack*, como dito no parágrafo anterior, faz parte do *heap*. Ela é criada — ou são criadas — no momento de carregamento do programa, e é usada para possibilitar que mudanças de contexto sejam feitas durante a execução do programa e a linha principal de execução consiga seguir seu fluxo normal mesmo assim. Tais "mudanças de contexto" podem ser chamadas internas a funções, em linguagens como C; a métodos, em linguagens como Java, ou mesmo chamadas a outras rotinas de outros programas, externos ao programa inicialmente em execução.

Embora a *stack* esteja dentro do *heap*, sua alocação é estática, ou seja, não mudará durante a execução do programa. Basicamente, a *stack* tem a estrutura apresentada na imagem a seguir.

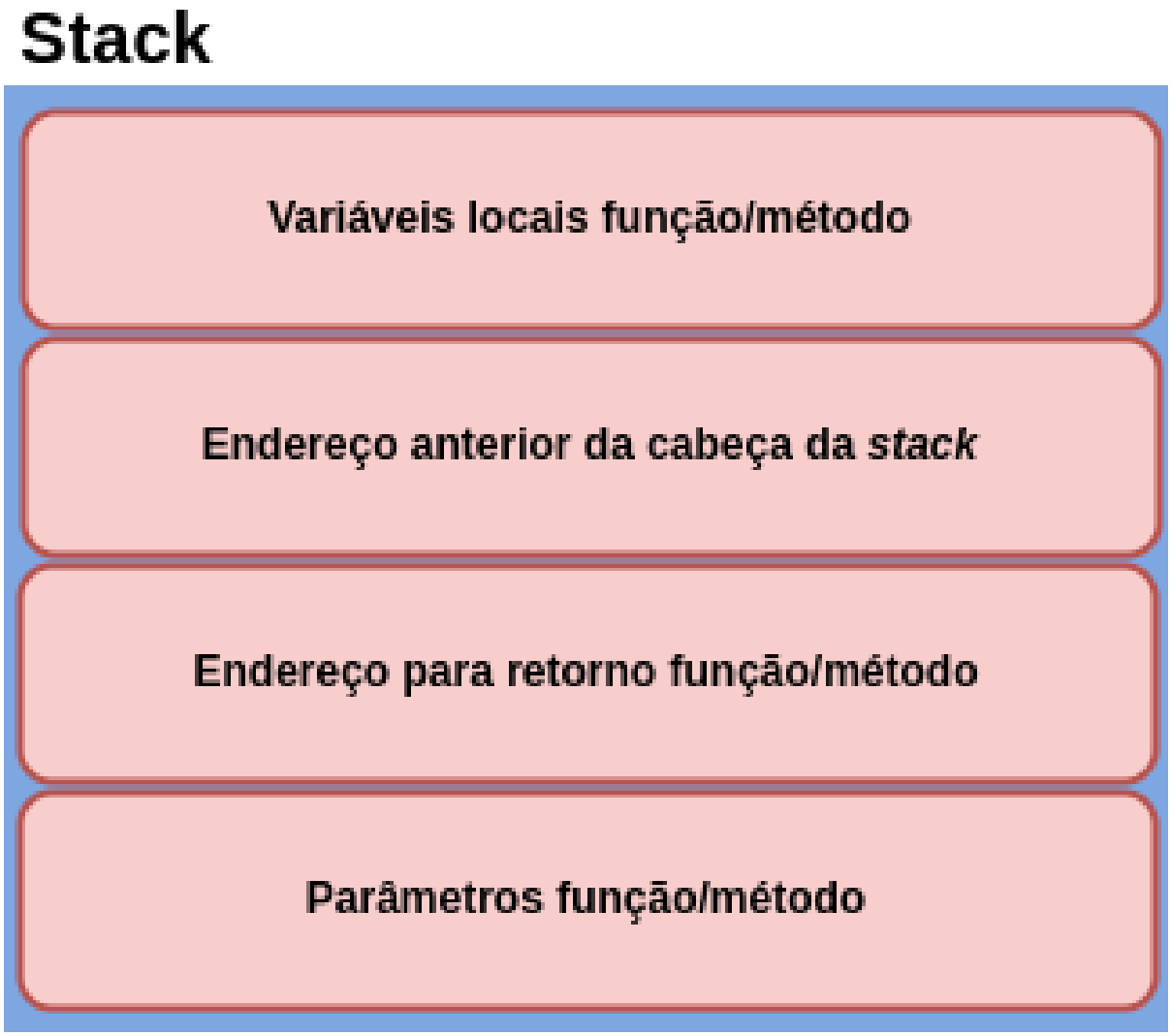


Figura 2.3: Modelo conceitual de uma stack.

Dessa forma, se várias funções ou métodos forem chamados em sequência *uma dentro da outra*, a estrutura anterior ficará *uma em cima da outra*, como na imagem a seguir.

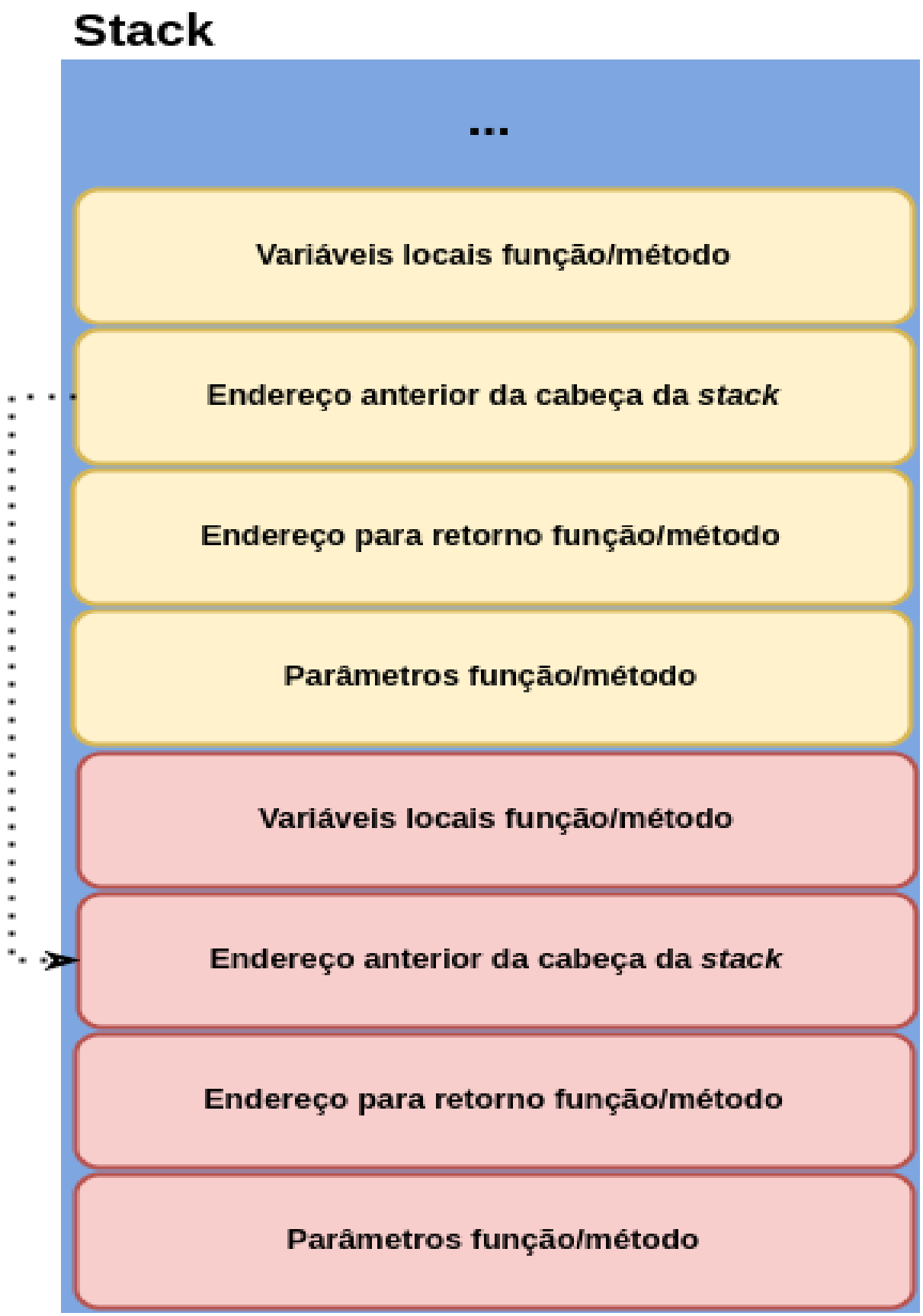

Stack
...
Variáveis locais função/método
Endereço anterior da cabeça da stack
Endereço para retorno função/método
Parâmetros função/método
Variáveis locais função/método
Endereço anterior da cabeça da stack
Endereço para retorno função/método
Parâmetros função/método

Figura 2.4: Modelo conceitual de stack de chamadas a funções/métodos.

Caso essa sequência de chamadas não seja *uma dentro da outra*, a *stack* vai colocando e retirando cada uma das chamadas. Ou seja, a *stack* sempre ficaria como na figura 2.3.

Para ilustrar melhor como *heap* e *stack* funcionam em conjunto, vamos ver um exemplo mais detalhado. Para isso, levemos em consideração o código Java a seguir, que pode ser encarado como um "fictício programa completo". Esse programa simplesmente exibe o valor do atributo `numero` para cada objeto criado a partir da classe `Classe2` e soma o conteúdo desses atributos. Tais objetos estão armazenados em um vetor de duas posições.

```
class Classe1 {

   Classe2[] classes = new Classe2[] {new Classe2(), new Classe2()};

   void listarValores() {

      for(Classe2 c: classes) {
         System.out.println(c.valor);
      }

      System.out.println("A soma dos valores é " + somarValores("xpto",
2.5));
   }

   int somarValores(String param1, double param2) {

      int total = 0;
      for(Classe2 c: classes) {
         total += c.numero;
      }

      return total;
   }

}
```

```
class Classe2 {

   int numero = 2;

}

public class Main {

   public static void main(String[] args) {

      Classe1 c1 = new Classe1();
      c1.listarValores();
   }
}
```

Na imagem a seguir, que simula a execução do programa de exemplo, percebe-se que o programa foi carregado no *heap* e, nele, as *stacks* necessárias foram criadas durante a execução do programa, assim como os objetos e vetores necessários. Na *stack*, as variáveis locais a cada método foram alocadas e os apontamentos foram feitos para os objetos do *heap* que os métodos usam. Também foi feito o encadeamento das chamadas na *stack*.

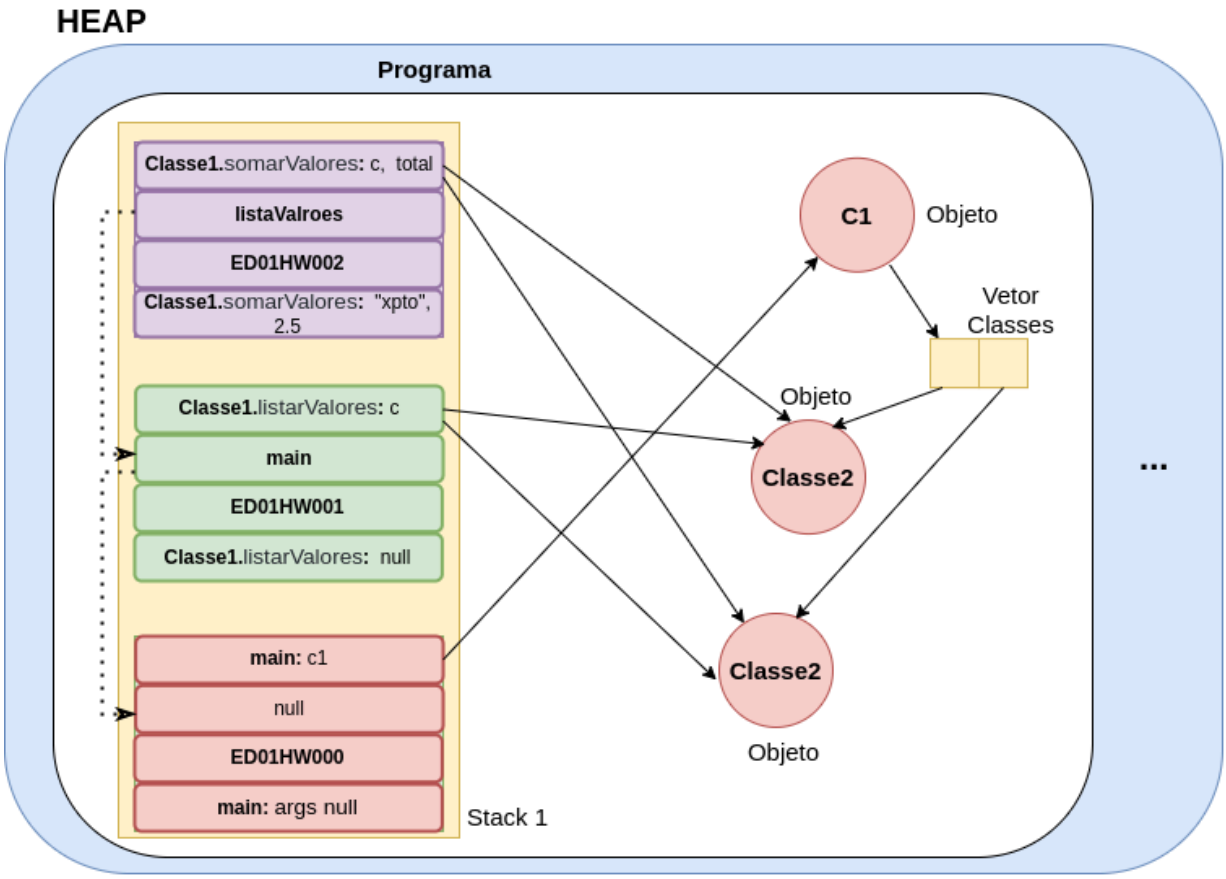


Figura 2.5: Heap e Stack funcionando em conjunto.

Por fim, é válido dizer que, quando o programa terminar, tudo isso será descarregado da memória, que ficará livre para outro programa usar. Outra observação final e importante é que, embora conceitualmente apresentemos o *heap* e *stack* em "caixinhas organizadas", na verdade os dois são criados de forma espaçada (desorganizada) na memória. Entretanto, o SO tem a capacidade de nos disponibilizar essas caixinhas, para sua compreensão e uso serem mais fáceis.

2.8 Resumo

Após esses conceitos básicos e algumas categorizações, podemos apresentar duas imagens que resumem o que foi exposto aqui. Nesta primeira imagem, levamos em consideração como os tipos de dados podem ser constituídos:

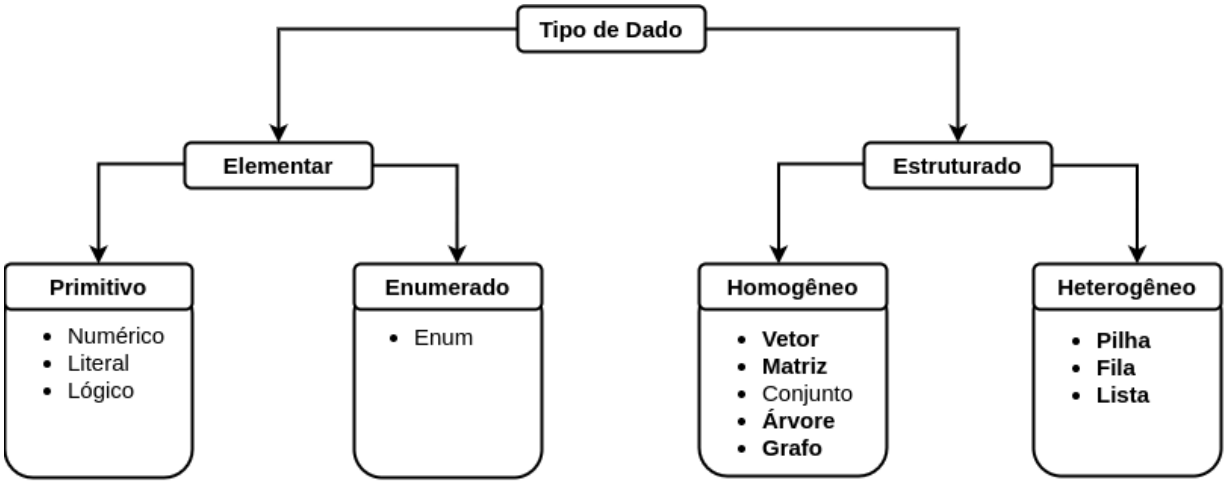


Figura 2.6: Constituição de TDs.

Já nesta segunda imagem levamos em consideração como os dados podem ser estruturados. É válido ressaltar que apenas os TADs são apresentados nesta imagem.

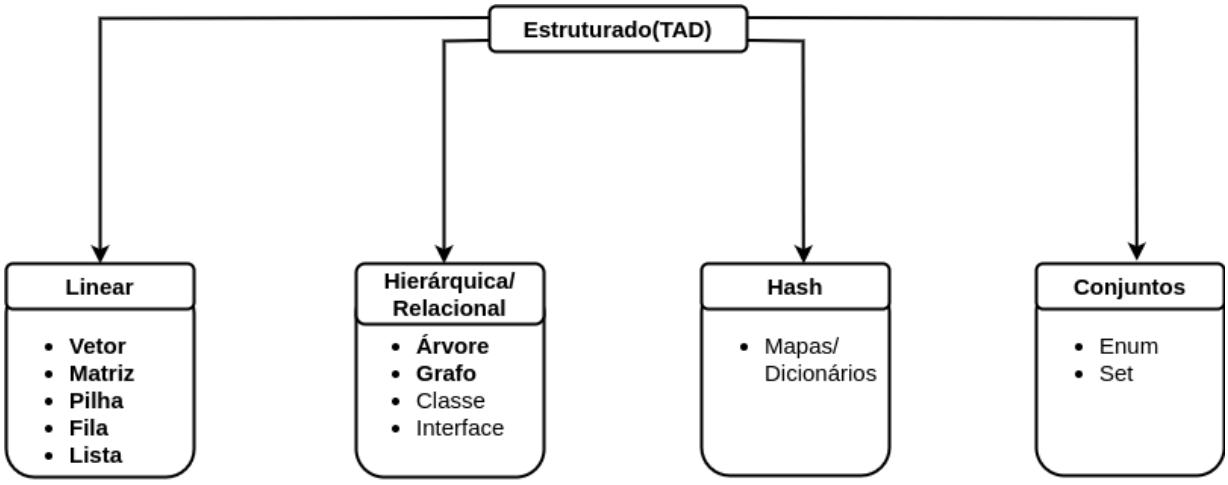


Figura 2.7: Estruturação de TADs.

Nota-se que em ambas as imagens há alguns termos em destaque. Esses termos são as principais EDs que serão abordadas neste livro.

Elas são o foco principal. As demais fazem parte de outras disciplinas no universo da programação e algumas serão apresentadas nos capítulos seguintes, mas de forma breve. Neste momento, podemos apenas falar um pouco de `Enum`, que não é de fato uma ED, mas tem um comportamento muito similar à ED *set* e por isso foi citado aqui. A maioria das linguagens de programação disponibilizam esse tipo e sugiro estudá-lo para melhor usá-las.

CAPÍTULO 3
Vetor e Matriz

Embora estas sejam as estruturas de dados mais básicas existentes e que, geralmente, são estudadas de forma inicial em disciplinas como algoritmos e lógica de programação, este capítulo apresenta algumas informações importantes e que nem sempre são exploradas quando o assunto é **vetor** e **matriz**. Além disso, serão apresentados códigos em C para a criação e uso dessas estruturas.

3.1 Fundamentos

Vetor e *matriz* são os termos em português utilizados para separar o que, em inglês, é conhecido como *array*. Essa estrutura pode ser unidimensional (vetor) ou multidimensional (matriz), e pode ser utilizada quando a quantidade de dados a ser armazenada for pequena e, principalmente, quando se sabe que essa quantidade é limitada e fixa. Também se pode levar em consideração se apenas um mesmo tipo de dados deve ser armazenado.

Embora existam essas diferenças em relação à quantidade de dimensões, existem algumas características comuns a vetores e matrizes.

Vetores e matrizes têm tamanho fixo

Vetores e matrizes têm um tamanho limitado, definido no momento de suas criações e que não mais poderá ser modificado.

Vetores e matrizes são sequenciais

Isso quer dizer que os elementos armazenados nessas estruturas estão dispostos de forma "uma após a outra" na memória do computador. Isso termina por criar duas características:

1. Acesso posicional: sempre acessamos os elementos através de suas posições dentro da estrutura;
2. Acesso rápido: para acessar um elemento, não é necessário passar pelos seus antecessores.

Homogêneo

Isso indica que tais estruturas só podem ser de um único *tipo de dado*: é um vetor de inteiros ou um vetor de caracteres; uma matriz de números reais ou uma matriz de números inteiros etc. Um determinado vetor ou uma determinada matriz nunca pode ter, em si, elementos de tipos de dados diferentes.

3.2 Vetores

Como citado, vetores são "arrays unidimensionais". Isso significa que ele é uma estrutura onde os elementos estão dispostos em um único sentido — por exemplo, horizontalmente. A figura a seguir ilustra isso.

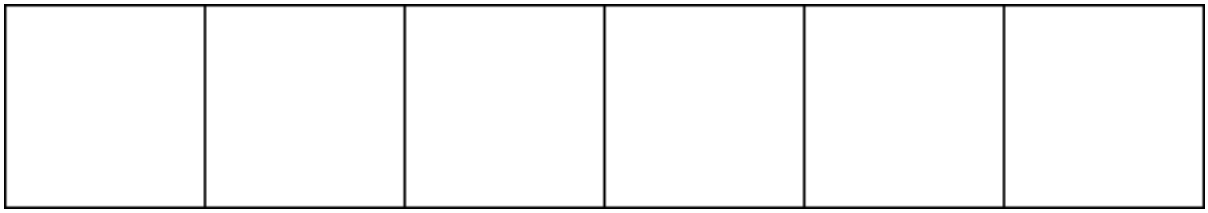


Figura 3.1: Modelo conceitual de um vetor.

Analisando essa imagem de forma mais minuciosa, algumas terminologias podem ser apresentadas:

1. Elemento: é o dado que está armazenado em algumas das posições do vetor;
2. Tamanho: é a capacidade total de elementos que o vetor pode armazenar;
3. Índice: é o número de uma das posições(limitada ao tamanho) que o vetor pode possuir.

A imagem a seguir ilustra um exemplo mais completo de um vetor.

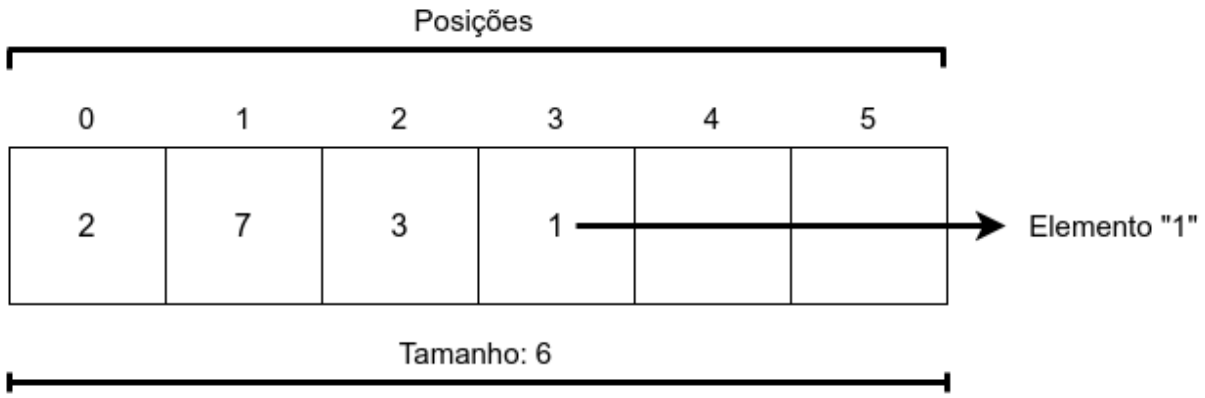


Figura 3.2: Vetor completo.

Nessa imagem, temos um vetor de tamanho 6, o qual tem seus índices (posições) variando de `0` até `5`. Existem apenas quatro elementos nesse vetor: `2`, `7`, `3` e `1`. Eles ocupam respectivamente as posições `0`, `1`, `2` e `3` do vetor. Nesse caso, as posições `4` e `5` estão vazias. Muitas linguagens são *zero-based*, o que significa que os índices do vetor começam em `0` (zero) e terminam em um dígito abaixo do tamanho total — no caso da imagem anterior, o tamanho é `6`, mas os índices vão de `0` até `5`.

Definição e uso

Para criar um vetor em C, deve-se aplicar a seguinte regra:

```
tipo_dado nome_vetor[tamanho];
```

Exemplo:

```
int idades[4];
```

No código anterior, foi alocado um espaço fixo de `4` posições e sequencial na memória para representar um vetor, que é acessado através do nome `idades`. Nesse espaço, podem-se armazenar inteiros (`int`) que conceitualmente representam idades. Para armazenar algum valor nas posições desse vetor, fazemos atribuições às suas posições. Aleatoriamente, o vetor foi iniciado da seguinte forma nas seguintes posições:

```
idades[0] = 10;
idades[1] = 15;
idades[3] = 8;
```

No código anterior, na posição `3` (índice 2), não foi armazenado nenhum valor ainda. Vale ainda ressaltar que, em C, *arrays* são *zero-based*. Para nosso vetor de idades, então, é de `0` até `3`. Por fim, para usar alguns dos elementos do vetor, usam-se suas posições:

```
int soma_idades = idades[1] + idades[3];
```

Como já tinha sido dito, o *acesso rápido* é decorrente da seguinte situação:

```
Elemento   idades[0]    idades[1]    idades[2]    idades[3]
Endereço   Edx00451     Edx00452     Edx00453     Edx00454

printf(idades[3]);
```

Nesse código, é ilustrado que o vetor `idades` — hipoteticamente — foi criado a partir do endereço `Edx00451` até o endereço `Edx00454`. Como vetores são estruturas sequenciais e inteiros têm um tamanho fixo, o acesso ao elemento na posição 4 (`idades[3]`) é direto. Isso é possível devido a uma conta que pode ser realizada para isso:

tamanho(tipo) * posicao_matriz

Nesse caso, o elemento na posição `4` seria:

```
tamanho(int) * 4
```

Dessa forma, ao usarmos o elemento que está na posição `4` desse vetor, conseguimos ir diretamente a ele, "pulando" seus elementos antecessores (índices `0`, `1`, `2`).

3.3 Matrizes

Como citado, matrizes são "arrays multidimensionais". Isso significa que matriz é uma estrutura onde os elementos estão dispostos em vários sentidos, por exemplo horizontalmente e verticalmente. Nesse caso, temos uma matriz bidimensional, que é a mais comum, mas matrizes podem ter quantas dimensões forem necessárias. A figura a seguir ilustra uma matriz bidimensional.

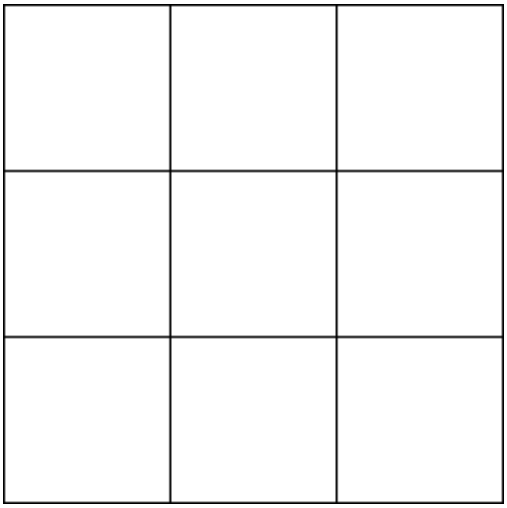


Figura 3.3: Modelo conceitual de uma matriz.

Mais uma vez, algumas terminologias podem ser apresentadas:

1. Elemento: é o dado que está armazenado em algumas das posições da matriz;
2. Tamanho: é a capacidade total de elementos que a matriz pode armazenar;
3. Índices: são os números de uma das posições (limitada ao tamanho) que a matriz pode possuir.

A imagem a seguir ilustra um exemplo mais completo de uma matriz.

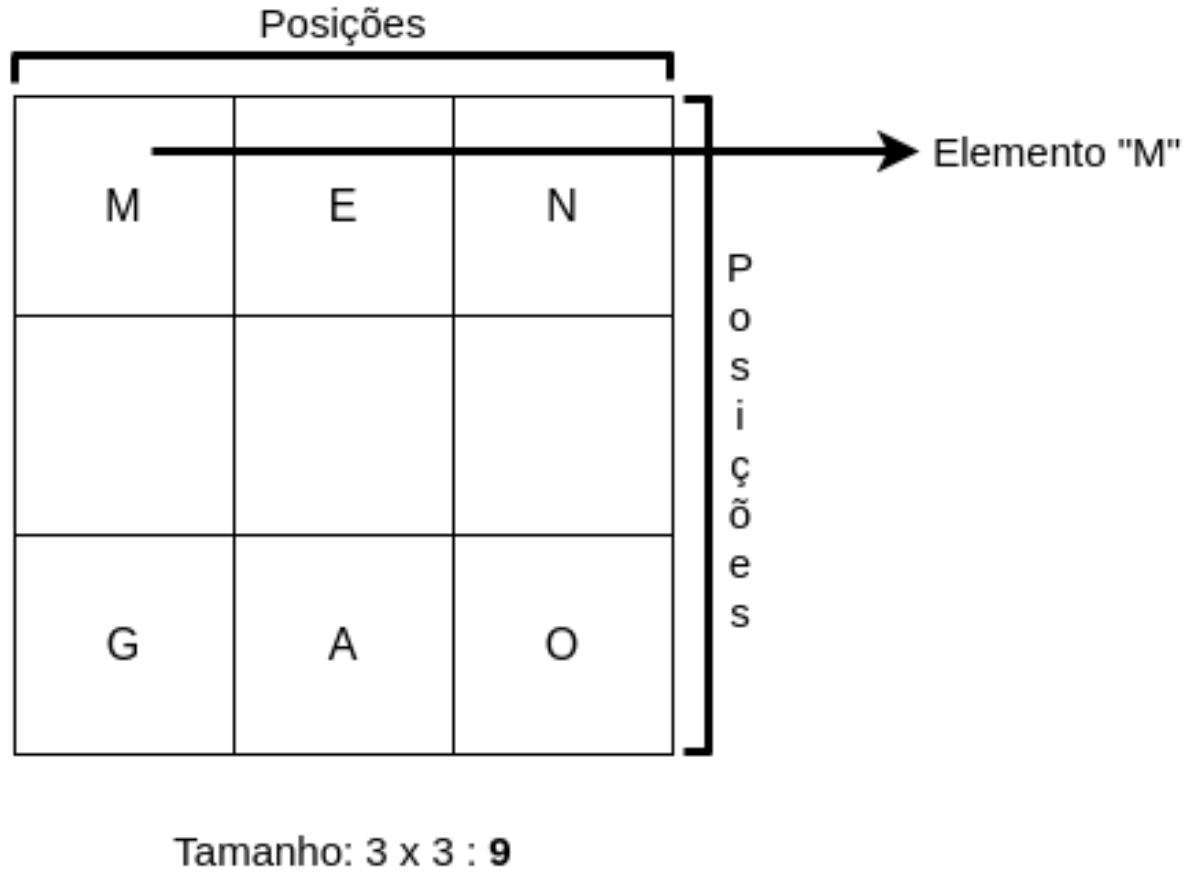


Figura 3.4: Matriz completa.

Nessa imagem, temos uma matriz de tamanho 3x3. Seus índices (posições) são combinações, que vão variando de acordo com suas dimensões, no caso `[0][0]`, `[0][1]`, `[0][2]`, `[1][0]`, `[1][1]`, até

chegar a `[2][2]`. Existem apenas 6 elementos nessa matriz: `M`, `E`, `N`, `G`, `A` e `O`. Eles ocupam as posições `[0][0]`, `[0][1]`, `[0][2]`, `[2][0]`, `[2][1]` e `[2][2]`. Nesse caso, as demais posições estão vazias.

Embora conceitualmente representemos matrizes como na figura 3.3, essa representação apenas é usada para torná-la mais compreensível. Na verdade, na memória do computador, as matrizes são criadas como vetores dentro de vetores. A imagem a seguir ilustra isso:

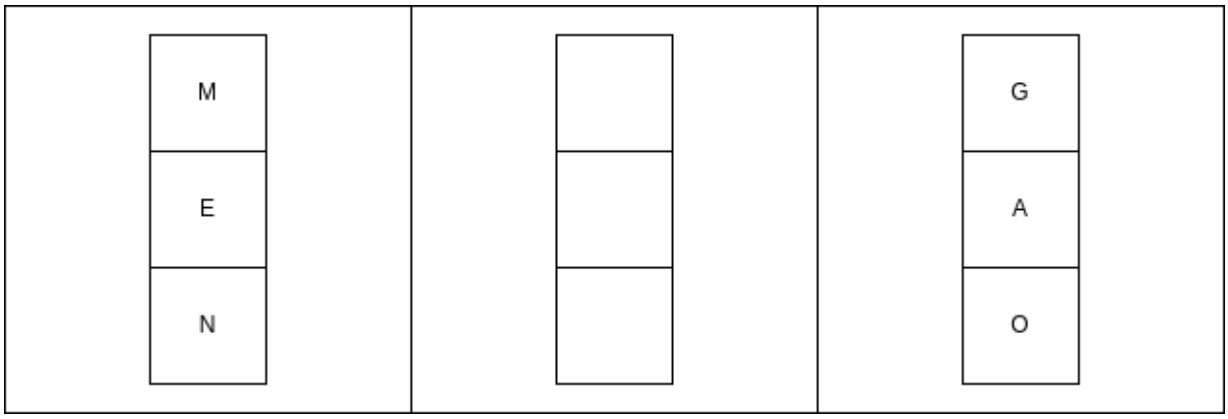


Figura 3.5: Modelo real de uma matriz.

Por fim, é válido ressaltar que tudo o que foi dito para matriz bidimensional é aplicável à tridimensional, quadrimensional etc. Apenas aumentam a quantidade de dimensões e, consequentemente, de índices, e a profundidade dos vetores dentro de vetores. Esses casos são mais raros e específicos e também um pouco mais complexos de criar e manusear, mas, se necessário, podem ser utilizados.

Outra observação pertinente é que a matriz não é obrigada a ser sempre simétrica — 3x3, 3x3x3, 5x5, 5x5x5, 2x2x2x2 etc. É possível, mas também não usual, termos matrizes irregulares, como 4x3, 2x4x3 etc.

Definição e uso

Para criar uma matriz em C, deve-se aplicar a seguinte regra:

```
tipo_dado nome_matriz[tamanho][tamanho];
```

Exemplo:

```
char texto[3][3];
```

No código anterior, foi alocado um espaço fixo de 9 posições e sequencial na memória para representar uma matriz, que é acessada através do nome `texto`. Nesse espaço podem-se armazenar caracteres que conceitualmente representam letras. Para armazenar algum valor nas posições dessa matriz, fazemos atribuições às suas posições:

```
texto[0][0] = 'M';
texto[0][1] = 'E';
texto[0][2] = 'N';
...
```

Assim como os vetores, matrizes em C são *zero-based*. O código anterior ilustra esse comportamento. Por fim, para usar alguns dos elementos da matriz, usamos suas posições:

```
char time[] = {texto[0][0],texto[0][1],texto[0][2],texto[2][0],texto[2]
[1],texto[2][2]};
```

Tudo que foi dito sobre o *acesso rápido* para vetores vale também para as matrizes, apenas deve-se levar em consideração que são vetores dentro de vetores.

3.4 Exercícios

1) Crie um vetor de inteiros (`int`) de 10 posições. Preencha-o com os valores `10`, `20`, `30`, `40`, `50`, `60`, `70`, `80`, `90` e `100`. Use um `for` para exibir os valores deste vetor.

2) Crie uma matriz de caracteres (`char`) de 16 posições (4x4). Preencha-a com os valores `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`, `j`, `k`, `l`, `m`, `n`, `o` e `p`. Use dois `for` para exibir os valores desta matriz.

3) Faça um programa com um vetor com 10 elementos inteiros positivos (aleatórios e de sua escolha). Permita que seja solicitado um determinado valor inteiro e positivo para ser procurado neste vetor. Caso exista, o programa deve exibir o índice no qual o valor está posicionado. Caso contrário, o programa deve informar que o elemento não existe no vetor.

4) Faça um programa com uma matriz com 9 elementos (3x3) reais positivos (aleatórios e de sua escolha). Calcule e exiba a soma dos elementos de cada linha desta matriz.

5) Faça um programa com uma matriz 5x5 de inteiros positivos ou negativos (aleatórios e de sua escolha) e encontre e exiba o maior elemento desta matriz.

Capítulo 4
Pilha

Embora tenhamos falado inicialmente sobre *vetor* e *matriz*, a **pilha** (do inglês *stack*) pode ser considerada de fato a primeira estrutura de dado, pois ela contém um conjunto de operações, comportamentos e características bem mais meticulosos que as estruturas do capítulo anterior.

O termo *stack* já foi abordado anteriormente. No Capítulo 2, *Conceitos Básicos*, falamos sobre *heap* e *stack*. A *stack* que foi abordada anteriormente é a *pilha*, que agora vamos estudar mais profundamente. Lá, mostramos que ela pode ser usada para armazenar o contexto de execução de programas, mas o que a torna mais adequada para isso é o que veremos agora, junto de outros conceitos-chaves.

4.1 Fundamentos

Podemos definir uma *pilha* como:

> Uma estrutura de dados onde os elementos são dispostos, conceitualmente, um em cima do outro.

Embora isso possa, inicialmente, parecer muito abstrato, nos deparamos com pilhas constantemente no nosso dia a dia — por exemplo: pilhas de livros a serem arrumados em uma estante; pilhas de caixas em um estoque; pilhas de pratos a serem lavados etc. Ou seja, essa ED é visivelmente comum, e nos computadores não é diferente. Pode-se até mesmo dizer que ela é uma das principais EDs no que diz respeito ao "funcionamento interno" dos

computadores. Vamos começar a entender os motivos disso a partir de agora.

Figura 4.1: Pilhas.

Na definição anterior, o ponto crucial é a palavra *conceitualmente*. Embora internamente, na memória dos computadores, a pilha seja de fato uma estrutura linear, pois é constituída de elementos que são ligados através de seus endereços de memória, sejam eles de forma contígua ou não, quando falamos *um elemento em cima do outro*, criamos um conjunto de comportamentos e operações que definem de forma única o que a pilha é.

Para começar a destrinçar a pilha, podemos explorar o exemplo citado da "pilha de caixas". Sabemos que a *pilha de caixa* vai aumentando toda vez que uma nova caixa é colocada sobre a última anteriormente colocada na pilha. Também sabemos que, para retirar as caixas mais abaixo da pilha, temos que primeiro retirar as que estão sempre na parte de cima — afinal, se retirarmos uma do "meio da pilha", é certo que um acidente acontecerá.

Tomando como premissa o parágrafo anterior, podemos definir os comportamentos de uma pilha:

- LIFO (*last in, first out*): O último a entrar é o primeiro a sair; ou
- FILO (*first in, last out*): O primeiro a entrar é o último a sair.

Embora os comportamentos possam parecer diferentes, em sua essência terminam por prover um mesmo resultado final: os elementos de uma pilha são sempre acrescentados e retirados de um único ponto, no caso, do seu *topo*. O *topo* é sempre o elemento mais acima da *pilha*.

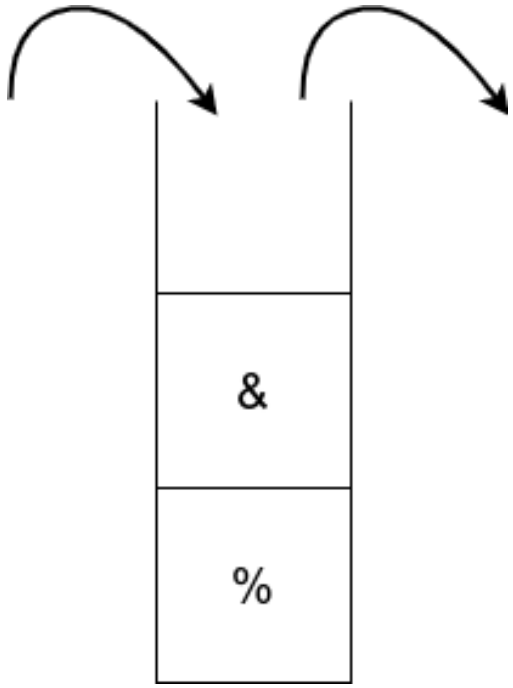


Figura 4.2: Pilha.

Na imagem anterior, a pilha possui dois elementos: `&` e `%`. Neste caso, `&` foi o último elemento a ser *empilhado* e `%`, o primeiro. Devido a isso, `%` só poderá ser *desempilhado* após `&` ser desempilhado primeiro. Em decorrência disso, o comportamento LIFO se torna mais comum para referenciar a pilha em relação ao FILO, embora este também seja válido.

4.2 Operações

Além dos comportamentos anteriormente apresentados, a organização "um em cima do outro" também provê operações que

podem ser realizadas sobre a pilha:

- `PUSH` : empilhar elemento;
- `SIZE` : retornar tamanho;
- `TOP` : retornar topo;
- `POP` : desempilhar elemento;
- `EMPTY` : verificar se está vazia.

A seguir, detalharemos tais operações, levando em consideração uma fictícia pilha, que se encontra inicialmente vazia.

PUSH

`PUSH` é a operação de empilhar elementos na pilha. Os novos elementos são empilhados no topo da pilha. Se tivermos, por exemplo, uma pilha chamada `P` e o elemento `$`, para que este seja empilhado, temos: `push(P,'$');`. A imagem a seguir mostra o resultado dessa execução.

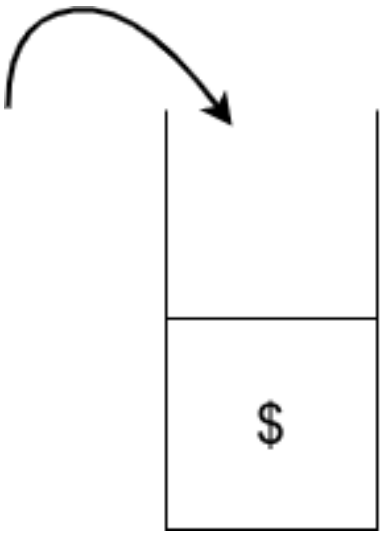


Figura 4.3: Pilha.

SIZE

`SIZE` é a operação que indica o tamanho atual da pilha. Esse valor pode aumentar ou diminuir à medida que elementos são adicionados ou retirados da pilha. Levando em consideração a pilha de exemplo `P`, temos: `size(P)`. O resultado dessa execução é `1`.

TOP

`TOP` é a operação que retorna o valor do topo da pilha, mas sem removê-lo. Dessa maneira, para a nossa pilha `P`, temos: `top(P);`. O resultado dessa execução é `$`.

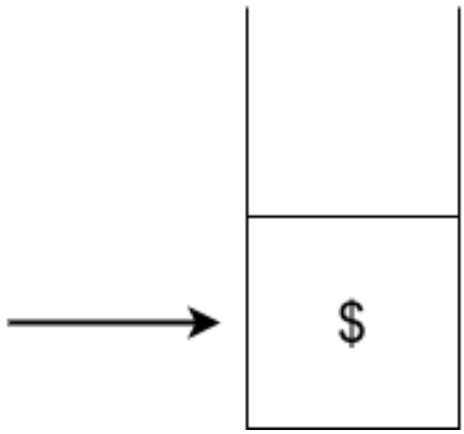


Figura 4.4: Pilha.

POP

`POP` é a operação de desempilhar elementos na pilha. Os elementos são desempilhados do topo da pilha e, nessa operação, o valor do elemento desempilhado deve ser retornado. Assim, na nossa pilha `P`, o desempilhamento seria: `pop(P);`. O resultado dessa execução é novamente `$`, mas desta vez este elemento é retirado da pilha.

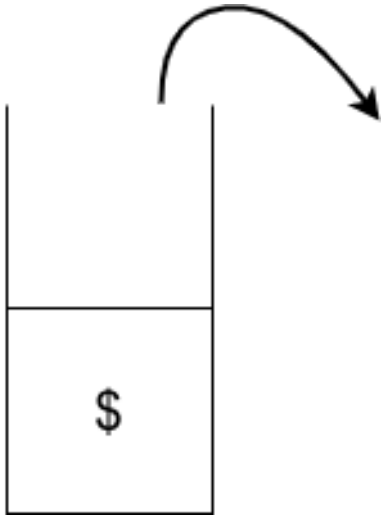


Figura 4.5: Pilha.

EMPTY

`EMPTY` é a operação que indica se a pilha está vazia, ou seja, se seu tamanho atual é 0 (zero) ou se ela se encontra ainda nula (`null`). Considerando novamente a nossa pilha de exemplo `P` , temos: `empty(P)` . O resultado dessa execução é `true` , pois o único elemento que tinha sido empilhado (`PUSH`), no caso `$` , foi desempilhado (`POP`).

4.3 Implementação

Nesta seção veremos os códigos em C para podermos realizar as operações da seção anterior, mas, antes, é válido ressaltar que pilhas podem ser implementadas com *vetores* ou *ponteiros*. A implementação usando *vetor* é mais simples e basta escolher o ponto que será o *topo* da pilha: o índice `0` ou o último índice do vetor, o qual representa seu tamanho máximo. Sugerimos como exercício que você mesmo realize essa implementação após a leitura desta seção ou ao final do capítulo.

Todavia, neste livro usaremos a implementação com *ponteiro*, que é um pouco mais complexa, mas é a mais utilizada, devido à capacidade de ter seu tamanho aumentado de acordo com a demanda. Nossa pilha será constituída de caracteres especiais, tais como `#`, `&`, `@`, entre outros.

Em virtude do que foi dito anteriormente acerca de nossa pilha, é necessário primeiro realizarmos dois passos:

1. Criar o registro (`struct`) que representa o valor e contém o ponteiro para os outros elementos da pilha, e criar a pilha em si, que será um conjunto de registros;
2. Iniciar a pilha.

Os códigos são, respectivamente:

```
typedef struct elemento {
 char valor;
 struct elemento *proximo;
} Elemento;

typedef struct pilha {
 Elemento *topo;
 int tamanho;
} Pilha;

Pilha* iniciar() {

 Pilha *p = malloc(sizeof(Pilha));
 p->topo = NULL;
 p->tamanho = 0;

 return p;
}
```

Nos dois primeiros códigos, o `struct` chamado `elemento` tem como finalidade aglutinar o dado que a pilha armazenará, assim como o ponteiro que será usado para conectar os elementos da pilha. Já o `struct` chamado `pilha` é responsável por definir a pilha de fato, que

é um encadeamento de vários *structs* do tipo `Elemento`. Além disso, esse `struct` também armazena o tamanho atual da pilha.

> Existem outras formas mais avançadas de aglutinar dados, mas isso requer um passo adiante: o estudo da Orientação a Objetos. Se tiver interesse, aconselho a leitura do livro *Orientação a Objetos: Aprenda seus conceitos e suas aplicabilidades de forma efetiva* (ISBN: 978-85-5519-213-5).

No terceiro trecho de código, a função `iniciar()` é responsável por alocar o espaço de memória para o primeiro elemento da pilha e iniciá-lo com `NULL`, pois nenhum elemento ainda foi empilhado. Além disso, esse código também configura o tamanho atual da pilha com `0`, pois ela acabou de ser criada. Ao final da configuração, a pilha é retornada e está pronta para começar a ser utilizada. Conceitualmente, a função `iniciar()` faz o que a imagem a seguir apresenta:

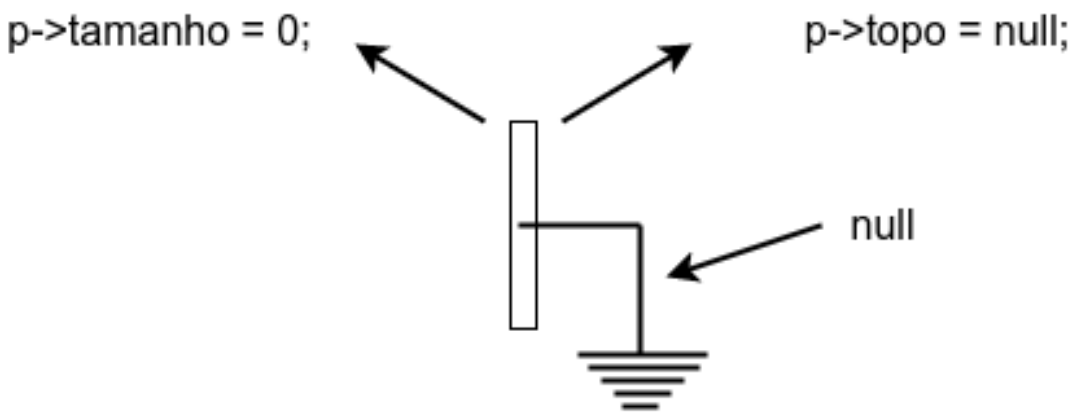


Figura 4.6: Pilha vazia.

No caso, somente cria e inicializa com `null` o "primeiro" elemento da pilha. Dessa forma, a pilha fica vazia, mas pronta para receber o primeiro elemento.

Para entender o que é um `struct` e o que faz a função `malloc`, aconselho a leitura dos apêndices II e III, respectivamente. De agora em diante, o uso desses conceitos de C se tornarão mais frequentes no livro.

Após a pilha ter sido definida e iniciada, vamos explorar as operações.

PUSH

```
void push(Pilha *p, char caractere) { /*I*/

 Elemento *e = malloc(sizeof(Elemento)); /*II*/
 e->valor = caractere; /*III*/
 e->proximo = p->topo; /*IV*/
 p->topo = e; /*V*/

 p->tamanho = p->tamanho + 1; /*VI*/
}
```

O código anterior, responsável por realizar a operação `push`, executa os seguintes passos:

I. Recebe como parâmetro a pilha na qual desejamos empilhar (`p`) um novo elemento e o valor a ser empilhado (`caractere`);

II. Aloca dinamicamente, com o auxílio da função `malloc`, um novo espaço de memória para armazenar um *struct* do tipo `Elemento`, que é responsável por guardar o valor a ser empilhado e apontar para o elemento mais antigo na pilha;

III. Coloca no campo `valor` o `caractere` a ser empilhado;

IV. Liga o novo `Elemento` ao antigo topo da pilha. Tal ligação é feita através de seus ponteiros;

V. Informa o que novo topo da pilha é o novo `Elemento` (`e`), que acaba de ser empilhado;

VI. Incrementa o tamanho da pilha em `1`.

Conceitualmente, após a execução do `push(p, '@')`, a pilha — que antes estava vazia — fica da seguinte forma:

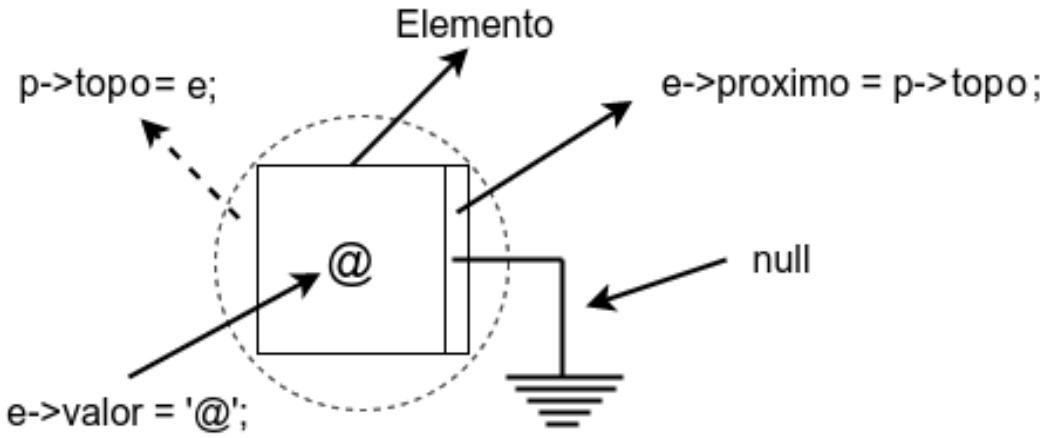


Figura 4.7: Demonstração do funcionamento de um empilhamento (push).

Podemos então pensar da seguinte forma: a cada vez que um novo elemento for empilhado, a imagem anterior vai aumentar para a esquerda, pois novos elementos serão conectados e, consequentemente, a pilha vai crescendo. A imagem a seguir ilustra isso. Tenhamos em mente que todos os itens da listagem de I até VI foram executados novamente. Pensemos também que as informações na figura 4.7 também são válidas. Uma observação é importante: temos que imaginar que o ponteiro do novo elemento aponta agora para o elemento que **era** o antigo topo da pilha.

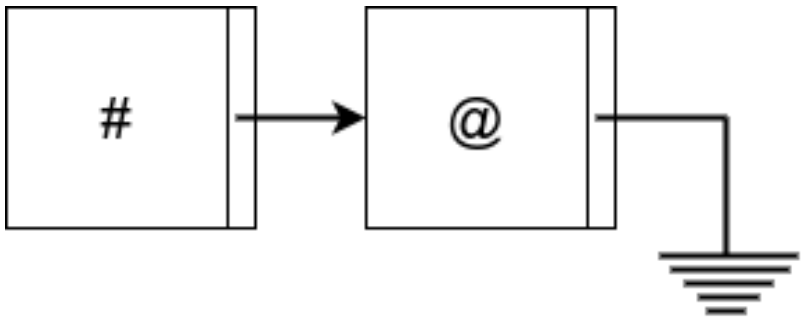


Figura 4.8: Visualização da pilha com ponteiros.

Por fim, podemos visualizar e pensar a nossa pilha da forma clássica:

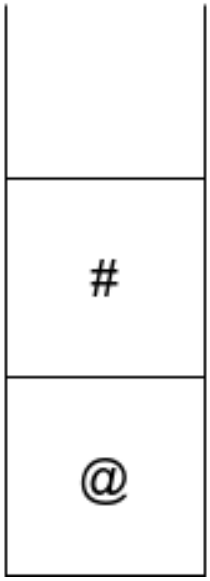


Figura 4.9: Visualização conceitual da pilha.

TOP

```
char top(Pilha *p) {

  if(empty(p)) { /*I*/

   printf("Pilha vazia!\n\n");
   return '\0';
  }

  return p->topo->valor; /*II*/
}
```

O código anterior, responsável por realizar a operação `top`, executa o seguinte passo:

I. Verifica se a pilha está vazia. Se sim, exibe a mensagem "Pilha vazia!" e retorna um `char` vazio (`\0`);

II. Caso contrário, retorna o valor do elemento da pilha `p` que é apontado pelo ponteiro que identifica o topo da pilha.

Levando em consideração que, além do `push(p, '@')`, também executamos o `push(p, '#')`, como exposto na seção anterior, a execução desse código exibe o caractere `#`.

SIZE

```
int size(Pilha *p) {
   return p->tamanho; /*I*/
}
```

O código anterior, responsável por realizar a operação `size`, executa o seguinte passo:

I. Retorna o tamanho da pilha `p`, que é armazenado no campo `tamanho` do *struct* `Pilha`.

Novamente, levando em consideração os dois `push` que foram realizados, a execução desse código exibe o valor `2`.

EMPTY

```
int empty(Pilha *p) {
   return p->tamanho == 0; /*I*/
}

int empty(Pilha *p) {
   return p->topo == null;
}
```

Os código anteriores podem ser responsáveis por realizar a operação `empty`. Ambos são válidos e surtem o mesmo efeito final: indicar se a pilha está vazia ou não. Levando em consideração o primeiro código, temos o seguinte passo:

I. Retorne o tamanho da pilha `p` e verifique se é igual a `0`. Se sim, a pilha está vazia e o valor `1` é retornado. Caso contrário, o valor `0`

é retornado.

Novamente, levando em consideração os dois `push` que foram realizados, a execução desse código exibe o valor `0`. Ou seja, a pilha não está vazia, em decorrência dos dois `push` realizados.

Todo o raciocínio que foi exposto para `p->tamanho == 0` se aplica a `p->topo == null`.

POP

```
char pop(Pilha *p) {

 Elemento *e;
 char c;

 if (!empty(p)) { /*I*/

  e = p->topo; /*II*/
  c = e->valor; /*III*/
  p->topo = p->topo->proximo; /*IV*/

  p->tamanho = p->tamanho - 1; /*V*/

  free(e); /*VI*/
  e = NULL;

  return c; /*VII*/
 } else {

  printf("Pilha vazia.\n\n");
  return '\0';
 }
}
```

Descartando a explicação das linhas `Elemento *e;` e `char c;`, que apenas são definições de variáveis, o código anterior é responsável por realizar a operação `pop` e realiza os seguintes passos:

I. Verifica se a pilha é vazia. Se for, exibe a mensagem "Pilha vazia." e finaliza a execução retornando um `char` vazio (`\0`). Caso contrário, segue ao passo II;

II. Armazena em `e` o topo atual da pilha;

III. Armazena em `c` o valor do elemento do topo da pilha;

IV. Faz o topo da pilha caminhar para o elemento abaixo dele, o qual será o *novo topo*;

V. Decrementa a quantidade de elementos da pilha;

VI. Libera o espaço de memória ocupado pelo antigo topo da pilha (`free(e);` e `e = NULL;`);

VII. Retorna o valor do antigo topo.

Por fim, levando novamente em consideração os dois `push` feitos anteriormente, o valor retornado pela execução do `pop` em nossa pilha é `#` , e agora o *novo topo* da pilha contém o valor `@` , que foi o primeiro valor a ser empilhado. Assim, nossa pilha fica — tanto logicamente como conceitualmente — da seguinte forma:

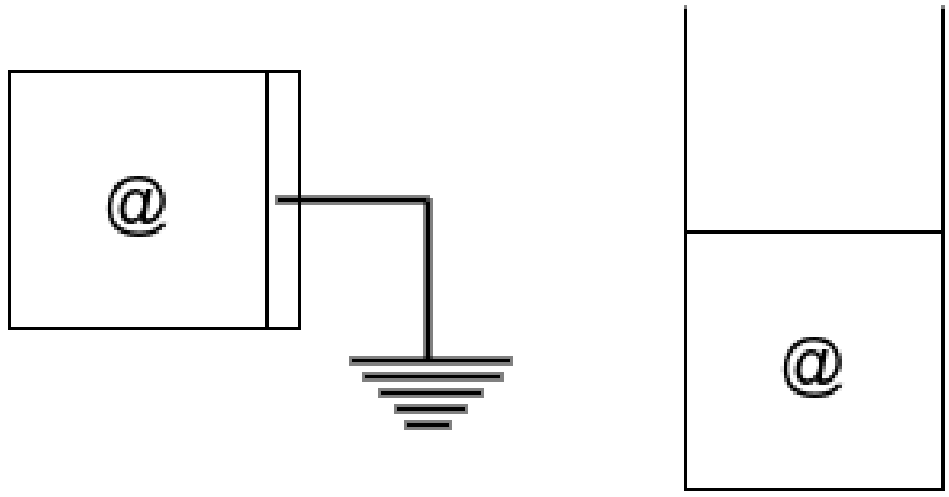


Figura 4.10: Pilha final.

Para entender o que faz a função `free`, aconselho, novamente, a leitura do apêndice III. Para obter o exemplo completo de nossa pilha em C, onde se pode executar (interagindo via prompt) e ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-Pilha.git. Lá você conseguirá fazer o download do código e executá-lo.

4.4 Exemplos de uso

Após a apresentação dos códigos e consequentemente uma melhor explicação de como uma pilha funciona de fato, uma pergunta pode surgir:

- Para que serve isso!?

Embora possa parecer algo "complexo e desnecessário", como já tinha sido dito antes, a pilha é uma das EDs mais utilizadas internamente pelos computadores. As seções a seguir listam alguns de seus usos.

Chamada de métodos

Esse é o exemplo de uso de pilha muito utilizado — indiretamente — por nós, programadores. Inclusive já falamos sobre ele na seção *Memória* do capítulo 2, mais precisamente *stack*, que em inglês significa *pilha*. A *stack* de execução é a pilha que estamos agora estudando. Toda vez que nossos programas chamam métodos, funções etc., o contexto da execução da aplicação e do método são armazenados (`push`) em pilhas. Assim, a aplicação tem acesso ao que precisa localmente para cada método e também sabe para onde deve retornar (`pop`) quando a execução atual se encerrar.

Um outro exemplo ainda relativo a chamadas de métodos são as **chamadas recursivas**, que é quando um método chama a si mesmo. No *Apêndice IV* explicamos mais detalhadamente o que é e como funcionam as chamadas recursivas, pois elas serão utilizadas por uma ED que apresentaremos mais adiante. Mas tenhamos em mente que elas não deixam de ser *chamadas de métodos*.

ctrl+z

Outro exemplo muito comum do uso de pilhas é o nosso famoso *desfazer*, ou, para os mais íntimos, `ctrl+z`. Quando estamos programando em uma IDE (Eclipse, IntelliJ, Visual Studio etc.), por exemplo, é muito comum darmos `ctrl+z` para retroceder algo digitado. Mais uma vez, uma pilha nos ajuda. As edições de nosso código-fonte vão sendo salvas (`push`) em uma pilha e, quando realizamos um `ctrl+z` (`pop`), o texto volta a um estado anterior.

Balanceamento de expressões

A forma mais prática de verificar se a seguinte expressão `A + (B * [C - D]) - {E / F}` está bem formatada é usando pilhas. Com elas, conseguimos verificar seu balanceamento, ou seja, se para cada `(`, `[` e `{` tem um `)`, `]` e `}`. Na seção *Exercícios* exploraremos isso.

Notações infixadas, prefixadas e pós-fixadas

Essas são as três formas que podemos realizar operações aritméticas. A mais difundida e usual é a *infixada*, mas as outras duas opções também são válidas. Os termos *pré*, *pós* e *in* indicam a posição dos operadores em relação a dois operandos. A notação prefixada possui o operador antes de dois operandos. Já na notação pós-fixada, o operador aparece após os dois operandos. Por fim, na notação infixada, o operador aparece entre os dois operandos. A lista a seguir exemplifica:

- Infixada: `1+2`, `(1+2)*(3-4)`
- Prefixada: `+12`, `*+12-34`

- Pós-fixada: `12+` , `12+34-*`

Nesse caso, podemos usar a pilhas para realizar conversões entre as notações ou para de fato executar o cálculo.

Esses foram apenas alguns de muitos exemplos de utilizações. O que temos que ter em mente é que pilhas são EDs muito úteis e entendê-las torna a manipulação de certos tipos de dados muito melhor.

4.5 Exercícios

> Todos os exercícios devem usar pilhas com ponteiros.

1) Levando em consideração uma pilha inicialmente vazia e que as seguintes operações foram realizadas, responda:

1. `pop()`
2. `push(p, 'a')`
3. `push(p, 'c')`
4. `top()`
5. `size()`
6. `push(p, 'g')`
7. `pop()`
8. `push(p, 'm')`
9. `pop()`
10. `top()`
11. `pop()`
12. `pop()`
13. `empty()`

a) Qual o retorno do passo 1?

b) Qual o retorno do passo 4?

c) Qual o retorno do passo 5?

d) Qual o retorno do passo 10?

e) Qual o retorno do passo 13?

2) Um palíndromo é uma palavra, frase ou número pode ser lido de forma igual em ambos os sentidos: da direita para a esquerda e da esquerda para a direita. Ex.: arara, 2002, "anotaram a data da maratona". Crie um programa em C que verifica se uma palavra, frase ou número é um palíndromo.

Dicas: Use uma pilha e faça o empilhamento e a verificação a cada desempilhamento com a palavra, frase ou número. Escolha entre palavra, frase ou número, pois as diferenças entre estes pode dificultar na resolução. Se quiser, faça um programa para cada.

3) Crie um programa em C que consiga determinar se uma expressão de entrada está balanceada.

Dica: Use uma pilha. Empilhe o oposto do símbolo a ser verificado e desempilhe quando forem iguais.

4) Dada a seguinte sequência de `push` em uma pilha, crie um programa em C que elimine o maior e o menor número.

1. `pilha.push(30);`
2. `pilha.push(10);`
3. `pilha.push(20);`
4. `pilha.push(50);`
5. `pilha.push(40);`

Dica: Use uma pilha auxiliar.

5) Este é um desafio. Considere uma pilha contendo os seguintes caracteres e nesta sequência de empilhamento: `@, &, @, #, %, &, $`. Faça um programa que use pilhas para eliminar os caracteres repetidos.

Dica: Use duas pilhas auxiliares.

CAPÍTULO 5
Fila

Neste capítulo, estudaremos a estrutura de dados **fila** (do inglês *queue*). Abordaremos seus conceitos-chaves, suas definições e mostraremos aplicações na computação e no nosso dia a dia. É válido ressaltar que as explicações de manuseio de ponteiros expostas no capítulo anterior também se aplicam neste capítulo sobre fila, portanto, não serão repetidas aqui tão detalhadamente.

5.1 Fundamentos

Podemos definir uma *fila* como:

> Uma estrutura de dados onde os elementos são dispostos de forma sequencial e ordenada.

À primeira vista, essa definição transparece que a fila é uma ED complexa, mas não é. Nós nos deparamos com filas todos os dias. Por exemplo: quando vamos ao supermercado, pegamos uma fila para pagar as compras; quando vamos abastecer o carro, podemos entrar em uma fila para chegar até a bomba de combustível para de fato abastecer. Enfim, existem várias outras situações nas quais podemos nos deparar com uma fila.

Assim como no nosso dia a dia, filas também são utilizadas na computação e se comportam da mesma maneira. Podemos até mesmo dizer que ela é uma ED que auxilia nossas atividades no dia a dia através de seu uso interno nos computadores. Vamos começar a entender mais sobre essa ED a partir de agora.

Figura 5.1: Fila.

Na definição anterior, o ponto crucial são as palavras "sequencial" e "ordenada". Internamente, na memória do computador, os elementos de uma fila são ligados através de seus endereços de memória, formando uma estrutura linear, a qual segue uma ordem bem definida de inserção e remoção de elementos. A existência de regras de inserção e remoção cria um conjunto de comportamentos e operações que definem de forma única o que uma fila é.

Para iniciar a desvendar a fila, podemos explorar o exemplo citado da "fila do supermercado". Sabemos que a fila do supermercado vai aumentando toda vez que uma nova pessoa chega e se posiciona ao seu final. Também sabemos que as pessoas vão sendo atendidas sempre do começo da fila até seu final. Ou seja, existe uma ordem no atendimento: quem chega primeiro é atendido primeiro e, consequentemente, sai primeiro. Não seria justo alguém que está no meio ou mesmo no final da fila ser atendido antes de quem está no começo da fila.

Tomando como premissa o parágrafo anterior, podemos definir os comportamentos de uma fila:

- FIFO (*first in, first out*): O primeiro a entrar é o primeiro a sair; ou
- LILO (*last in, last out*): O último a entrar é o último a sair.

Embora os comportamentos possam parecer díspares, em sua essência proveem um mesmo resultado final: os elementos de uma fila são sempre acrescentados em uma extremidade e retirados de outra. Isso permite duas abordagens:

1. Os elementos são acrescentados ao seu final e retirados de seu início, no caso, sua *cabeça*;
2. Os elementos são acrescentados em sua *cabeça* e retirados de seu final.

Ambas são abordagens válidas e resultam no mesmo comportamento, diferenciando-se apenas em um quesito referencial. Todavia, a primeira opção é a universalmente adotada.

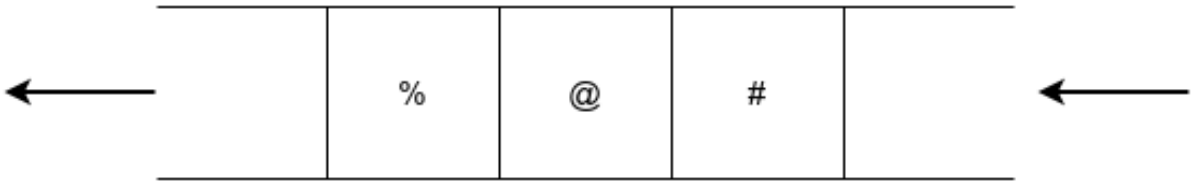


Figura 5.2: Representação conceitual de uma fila.

A imagem anterior apresenta uma fila que possui três elementos: `%`, `@` e `#`. Nesse caso, `%` foi o primeiro elemento a *entrar* na fila e `#`, o último. Consequentemente, `#` só poderá *sair* da fila após `%` e `@` saírem, e nessa ordem. Em decorrência disso, o comportamento FIFO se torna mais comum para referenciar a fila em relação ao LILO, embora este também seja válido.

5.2 Operações

Além dos comportamentos anteriormente apresentados, a organização "sequencial" e "ordenada" provê operações que podem ser realizadas sobre a fila:

- `ENQUEUE` : enfileirar elemento;
- `SIZE` : retornar tamanho;
- `HEAD` : retornar cabeça;
- `DEQUEUE` : desenfileirar elemento;

- `EMPTY` : verificar se a fila está vazia.

A seguir, detalharemos tais operações, levando em consideração uma fictícia fila, que se encontra inicialmente vazia.

ENQUEUE

`ENQUEUE` é a operação de enfileirar elementos na fila. No caso, os novos elementos são adicionados ao final da fila. Considerando que temos uma fila chamada de `F` e o elemento `$`, para que ele seja enfileirado, temos: `enqueue(F,'$');`. O resultado dessa execução está a seguir:

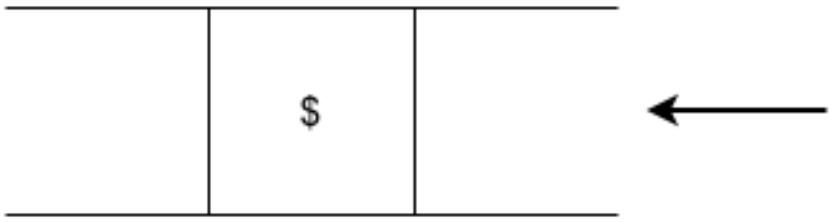


Figura 5.3: Fila após enqueue.

SIZE

`SIZE` é a operação que indica o tamanho atual da fila. Esse valor pode aumentar ou diminuir à medida que elementos são adicionados ou retirados da fila. Levando em consideração a fila de exemplo `F`, temos: `size(F)`. O resultado dessa execução é `1`.

HEAD

`HEAD` é a operação que retorna o valor da *cabeça* da fila, mas sem removê-lo. Dessa maneira, para a nossa fila `F`, temos: `head(F);`. O resultado dessa execução é `$`.

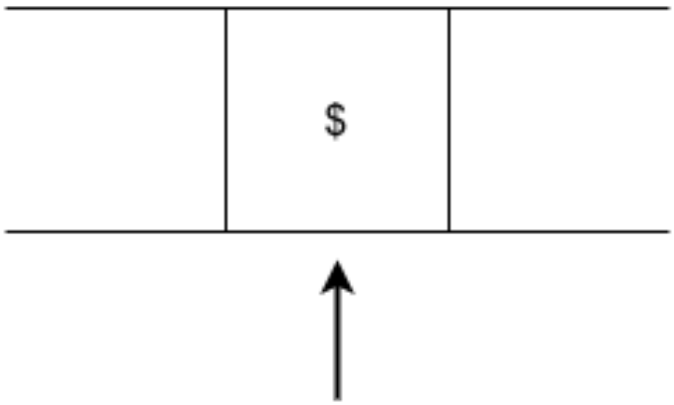


Figura 5.4: Cabeça da fila.

DEQUEUE

`DEQUEUE` é a operação de desenfileirar elementos na fila. No caso, os elementos são retirados da *cabeça* da fila e, nessa operação, o valor do elemento desenfileirado deve ser retornado. Dessa forma, tendo a nossa fila `F`, o desenfileiramento seria: `dequeue(P);`. O resultado dessa execução é novamente `$`, mas dessa vez esse elemento é retirado da fila.

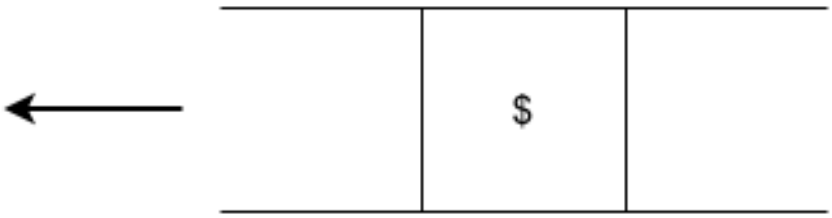


Figura 5.5: Fila após dequeue.

EMPTY

`EMPTY` é a operação que indica se a fila está vazia, ou seja, se seu tamanho atual é 0 (zero), ou se ela se encontra ainda nula (`null`). Novamente, levando em consideração a nossa fila de exemplo `F` , temos: `empty(F)` . O resultado dessa execução é `true` , pois o elemento `$` que havia sido enfileirado (`ENQUEUE`) foi desenfileirado (`DEQUEUE`).

5.3 Tipos de fila

Ao contrário da ED *pilha*, que era de certo modo mais simplista no que diz respeito a seu comportamento e uso, a ED *fila* possui mais de um tipo. Isso termina por prover mais comportamentos e formas de uso. Nas seções a seguir vamos apresentar esses tipos e seus comportamentos.

Clássica

A fila do tipo **clássica** é a fila mais comumente usada. Ela é basicamente a execução das operações anteriormente apresentadas sem nenhuma mudança em seus comportamentos. Tais operações podem ser usadas na ordem necessária e de acordo com a demanda.

Circular

A fila do tipo **circular** tem como principais características sua *cabeça* e *fim* "se tocarem" e os elementos ficarem "circulando" dentro da fila. Devido a isso, são criados controles para determinar onde está a cabeça (*head*) e o fim da fila. A imagem a seguir apresenta uma *fila circular* vazia.

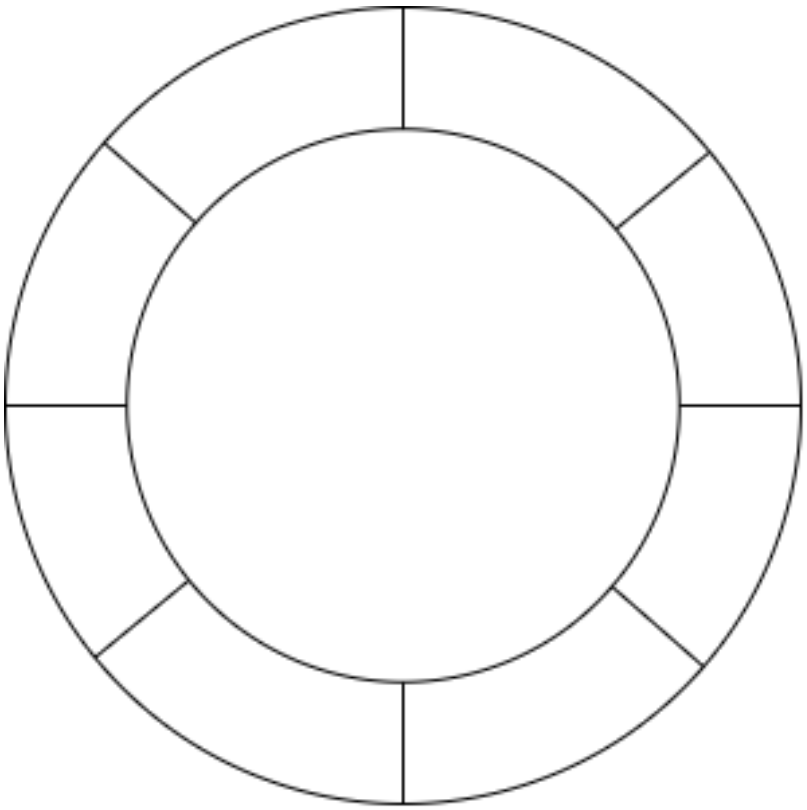


Figura 5.6: Fila circular.

Já na imagem a seguir, a fila teve a inserção do elemento `@`. Nesse caso, os controles indicam onde está a cabeça e o fim da fila agora.

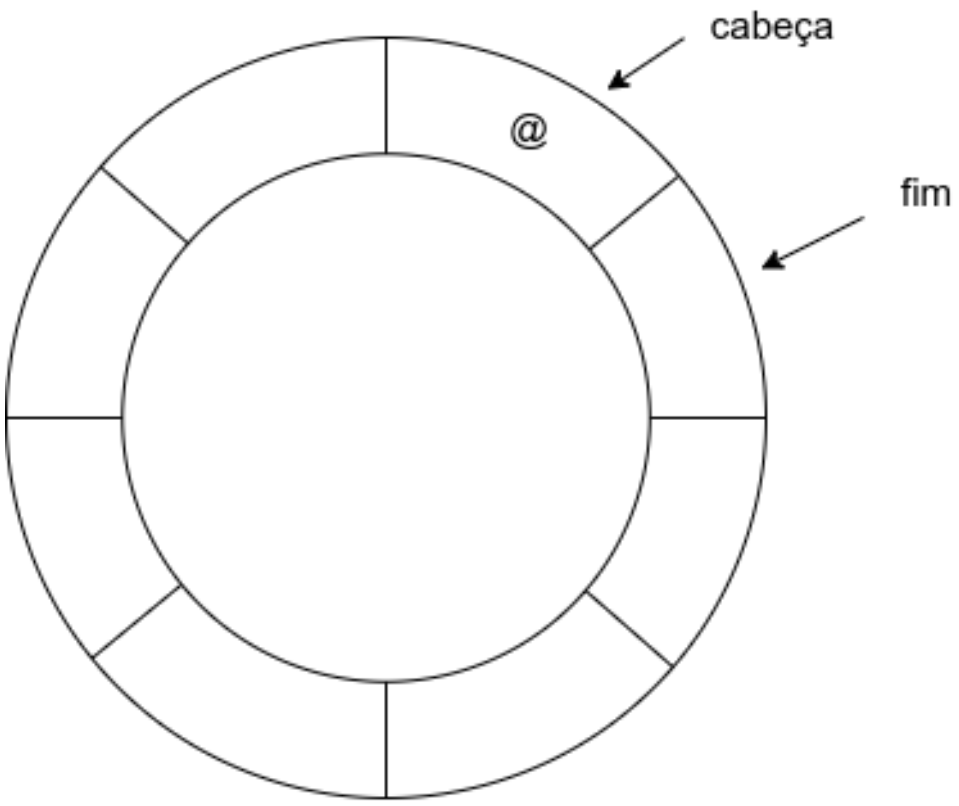


Figura 5.7: Fila circular com 1 elemento.

Na próxima imagem, a fila teve a inserção do elemento `#`, `$` e `&`. Além disso, o primeiro elemento enfileirado (`@`) foi desenfileirado. Nesse caso, os controles indicam onde estão a cabeça e o fim da fila agora.

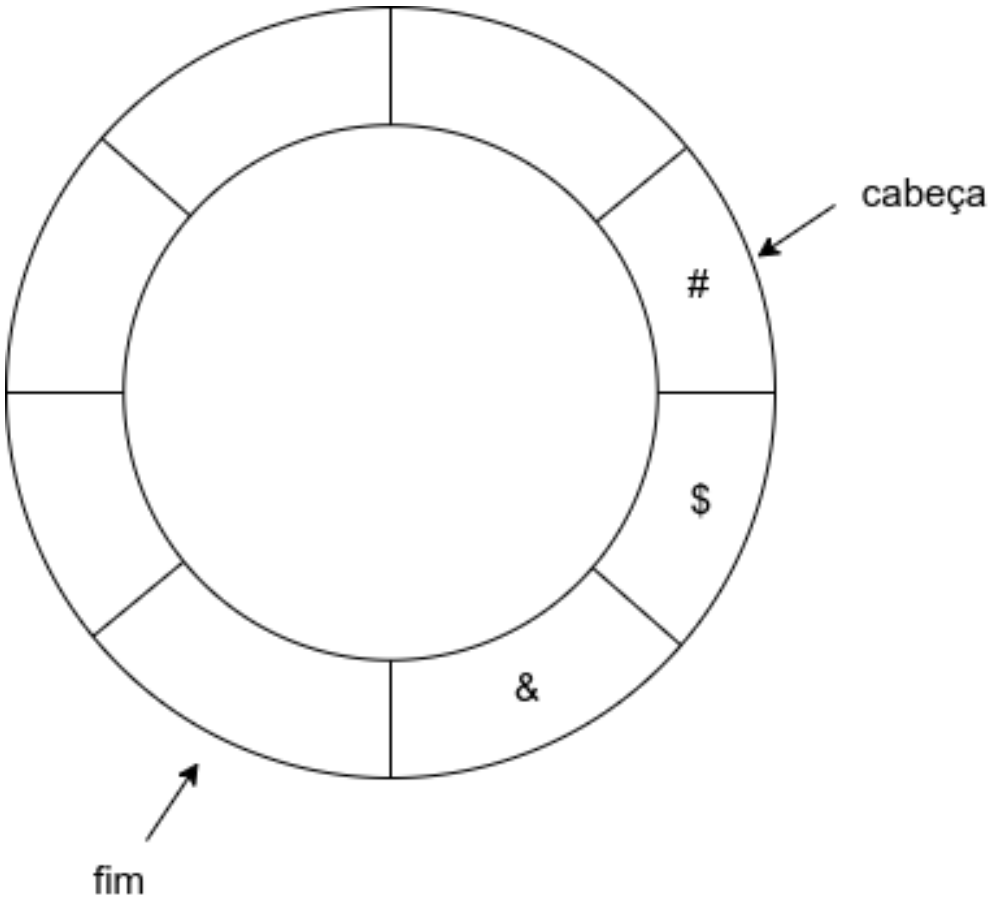


Figura 5.8: Fila circular após operações de enqueue e dequeue.

Por fim, podemos verificar que os controles ficam se movimentado entre os elementos da fila circular para indicar seu início e fim. É válido ressaltar que algumas vezes o fim estará antes do início, e é justamente por isso que a circularidade ocorre.

Prioridade

A fila do tipo **prioridade** possui modificações nas operações de enfileiramento (enqueue) e desenfileiramento (dequeue). Essas modificações alteram o comportamento padrão da fila, no caso o FIFO.

Em uma *fila de prioridade* onde o enfileiramento (enqueue) é modificado, temos o seguinte comportamento:

- Quando um novo elemento vai ser enfileirado, é verificada sua prioridade em relação aos elementos já existentes. A depender do tipo de prioridade (*maior* ou *menor*), esse novo elemento pode se posicionar ao final da fila ou ir "navegando" dentro da fila até encontrar sua posição adequada. Todavia, os desenfileiramentos (`dequeue`) são sempre na *cabeça* da fila.

Em uma *fila de prioridade* onde o desenfileiramento (`dequeue`) é modificado, temos o seguinte comportamento:

- Quando um elemento vai ser desenfileirado da fila, é verificada a prioridade de quem deve sair. Quem tiver a *maior* ou *menor* prioridade vai sair, mesmo que não se encontre na cabeça da fila. Todavia, os enfileiramentos (`enqueue`) são sempre no *fim* da fila.

Em ambas as modificações, citamos uma prioridade maior ou menor. Isso implica em dois tipos de prioridade:

1. Ascendente
2. Descendente

Para a fila de prioridade onde o enfileiramento (`enqueue`) é modificado, *ascendente* implica que a fila sempre estará ordenada da menor prioridade (cabeça) para a maior prioridade (fim). Já *descendente* implica que a fila sempre estará ordenada da maior prioridade (cabeça) para a menor prioridade (fim).

Já na fila de prioridade onde o desenfileiramento (`dequeue`) é modificado, *ascendente* implica que sempre será desenfileirado o elemento de *menor* prioridade. Já *descendente* implica que sempre será desenfileirado o elemento de *maior* prioridade. Nesse caso, a determinação de quem deve sair será obtida a partir de uma verificação que se inicia na cabeça da fila e vai até seu fim. Nesta abordagem, a fila não fica ordenada.

As imagens a seguir ilustram a execução de todos esses comportamentos.

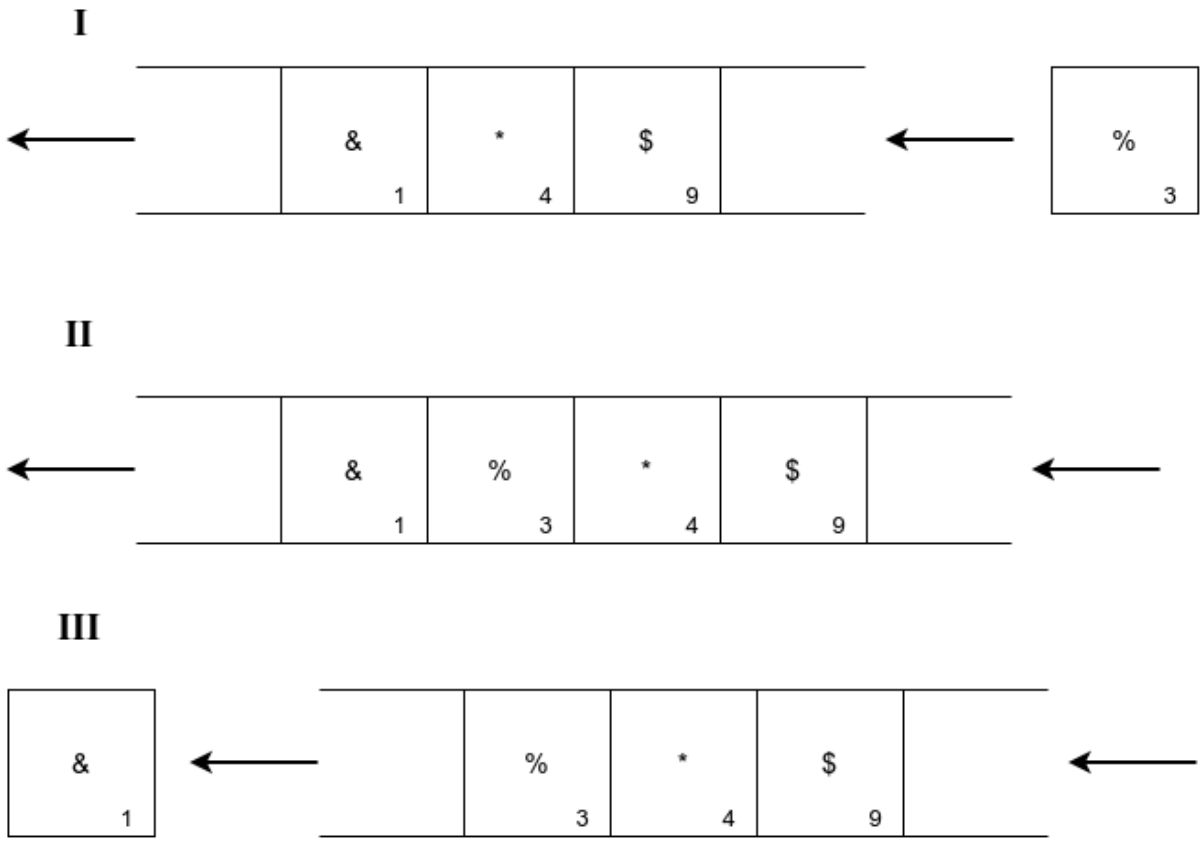


Figura 5.9: Fila de prioridade ascendente com enfileiramento modificado.

Na imagem anterior, temos uma *fila de prioridade ascendente com enfileiramento modificado*. Nesse caso, em I ela já possuía três elementos (`&` , `*` , `$`), cada um com as respectivas prioridades: 1, 4 e 9. Então, um novo elemento `%` , que tem a prioridade 3, será enfileirado (`enqueue`). Dessa forma, ele se posicionará na segunda posição da fila. O resultado disso é visualizado em II. Por fim, quando um desenfileiramento (`dequeue`) for executado, o elemento da *cabeça* da fila deve ser removido — nesse caso, o `&` . O resultado final é visto em III.

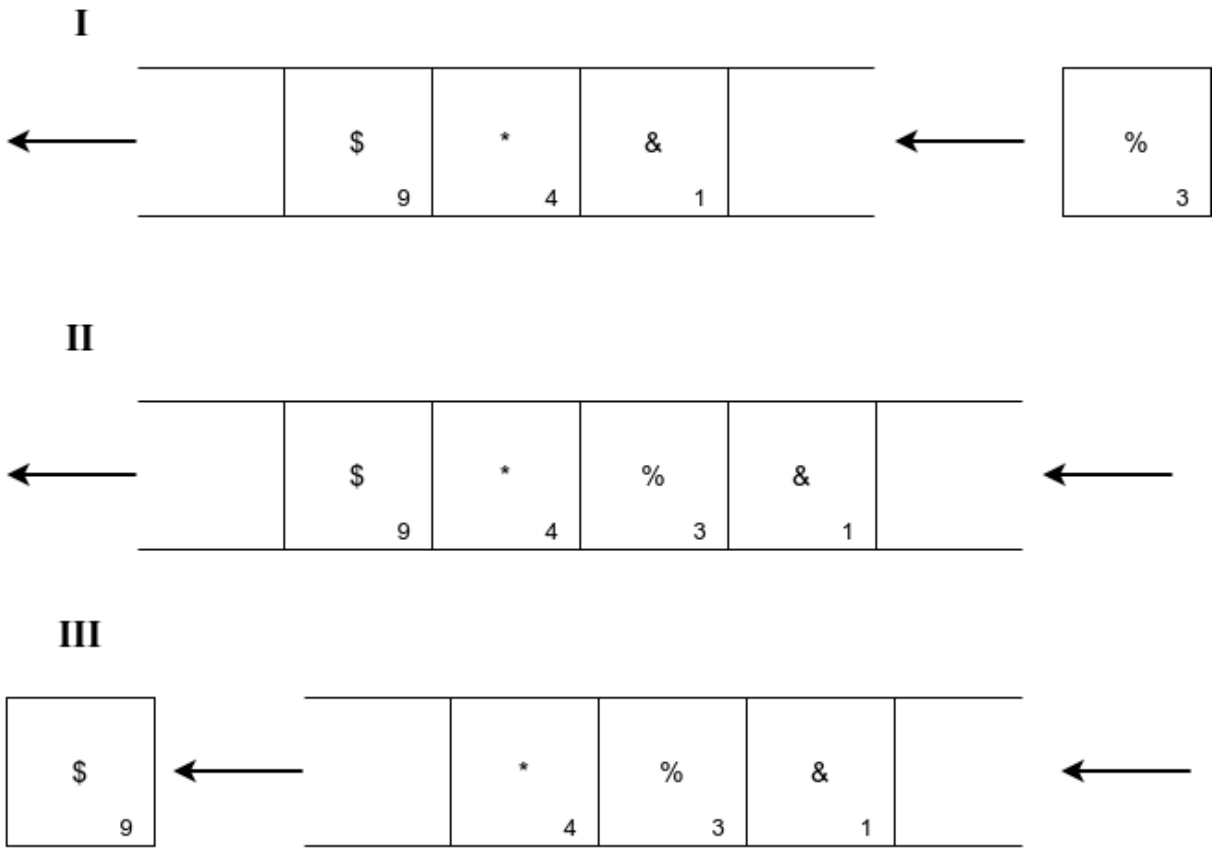


Figura 5.10: Fila de prioridade descendente com enfileiramento modificado.

Na imagem anterior, temos uma *fila de prioridade descendente com enfileiramento modificado*. Nesse caso, em I ela já possuía três elementos (`$` , `*` , `&`), cada um com as respectivas prioridades: 9, 4 e 1. Então, um novo elemento `%` , que tem a prioridade 3, será enfileirado (`enqueue`). Dessa forma, ele se posicionará na penúltima posição da fila. O resultado disso é visualizado em II. Por fim, quando um desenfileiramento (`dequeue`) for executado, o elemento da *cabeça* da fila deve ser removido — nesse caso, o `$` . O resultado final é visto em III.

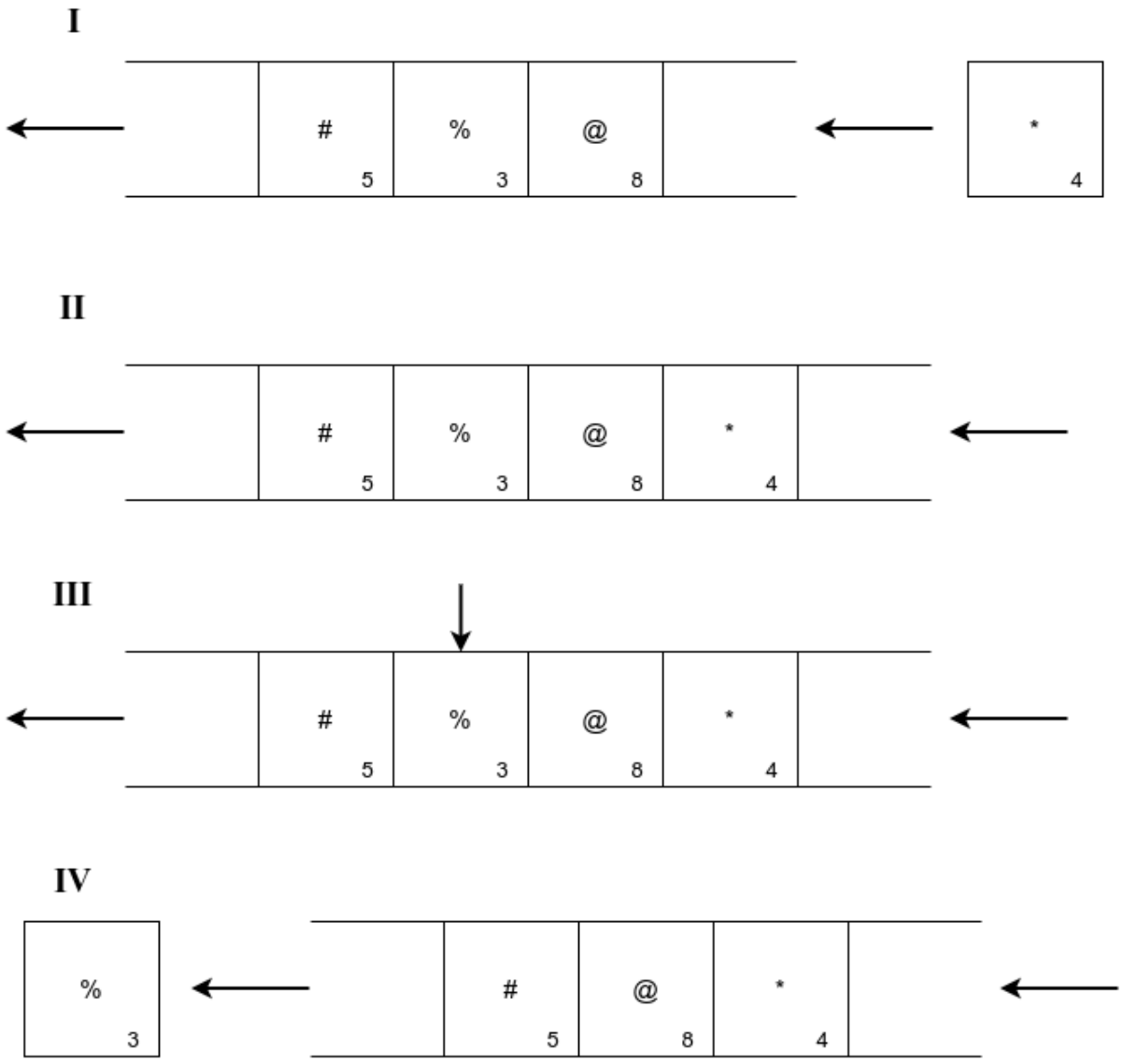


Figura 5.11: Fila de prioridade ascendente com desenfileiramento modificado.

Na imagem anterior, temos uma *fila de prioridade ascendente com desenfileiramento modificado*. Nesse caso, em I ela já possuía três elementos (`#` , `%` `@`), cada um com as respectivas prioridades: 5, 3 e 8. Então, um novo elemento `*` com prioridade 4 será enfileirado (`enqueue`). Dessa forma, ele se posicionará naturalmente na última posição da fila. O resultado disso é visualizado em II. Entretanto, quando um desenfileiramento (`dequeue`) for realizado nessa fila, será verificado quem é o elemento com a *menor* prioridade — nesse

caso, é o `%`, com prioridade 3, e o resultado disso é visualizado em III. Assim, este deverá ser desenfileirado, e o resultado é visualizado em IV.

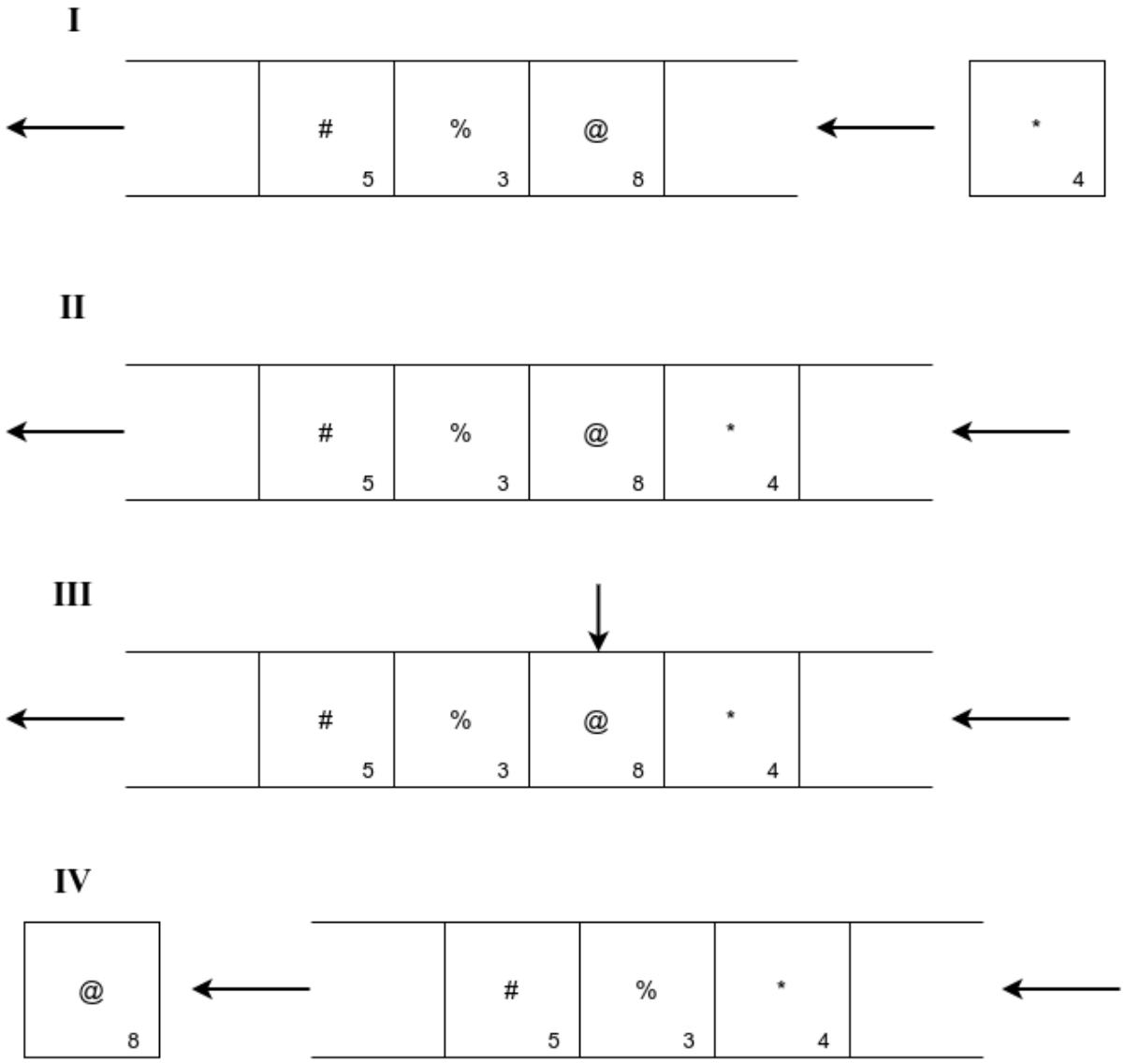


Figura 5.12: Fila de prioridade descendente com desenfileiramento modificado.

Na imagem anterior, temos uma *fila de prioridade descendente com desenfileiramento modificado*. Nesse caso, em I ela já possuía três elementos (`#`, `%` `@`), cada um com as respectivas prioridades: 5, 3 e 8. Então, o novo elemento `*` que possui a prioridade 4, será enfileirado (`enqueue`). Dessa forma, ele se posicionará naturalmente

na última posição da fila. O resultado disso é visualizado em II. Entretanto, quando um desenfileiramento (`dequeue`) for realizado nessa fila, será verificado quem é o elemento com a *maior* prioridade — nesse caso, é o `@` , com prioridade 8, e o resultado disso é visualizado em III. Assim, este deverá ser desenfileirado, e o resultado é visualizado em IV.

Com dois finais (Dupla, Deque, Dequeue, Com duas extremidades)

Um tipo muito peculiar de fila é conhecido como **fila com dois finais**. Nesse tipo de fila, é possível enfileirar (`enqueue`) e desenfileirar (`dequeue`) elementos em ambas as extremidades. A imagem a seguir demonstra isso.

Figura 5.13: Fila com dois finais.

A partir da imagem anterior, podemos ver que esse tipo de fila opera exatamente igual a uma fila *clássica*, mas para ambas as extremidades. Uma observação interessante: nesse tipo de fila, os elementos se organizam a partir do centro. Na prática, é como se duas filas se encontrassem, onde a *cabeça* de uma toca o *fim* da outra. É como se logicamente existissem duas filas, mas operacionalmente só uma. A imagem a seguir demonstra isso.

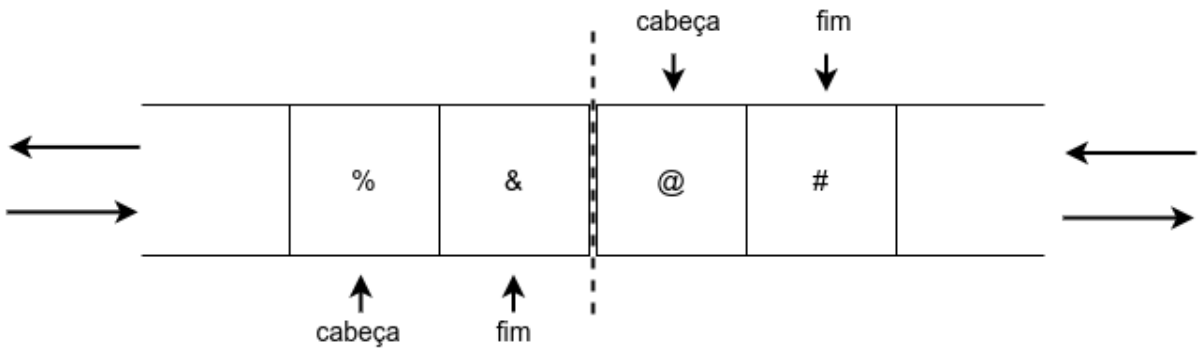


Figura 5.14: Esquema conceitual de fila com dois finais.

Duas observações relevantes a respeito de filas com dois finais são:

- Embora as imagens anteriores apresentem uma representação linear da mesma, esse tipo de fila sempre deve ser implementado de forma circular (similar ao tipo **circular**);
- Sempre deve ter uma capacidade máxima e fixa definida, de preferência par.

Para finalizar, os enfileiramentos e desenfileiramentos sempre ocorrem do mesmo lado. Isto é, se um elemento foi enfileirado pela "esquerda" da fila, deverá ser desenfileirado também pela "esquerda" e o mesmo vale para a "direita". Todavia, os enfileiramentos e desenfileiramentos podem ocorrer de forma paralela e dessincronizada. Caso necessário, esse tipo de fila também pode trabalhar com prioridades.

5.4 Implementação

Nesta seção, veremos os códigos em C para podermos realizar as operações da seção *Operações*. É válido ressaltar que filas podem também ser implementadas com vetores ou ponteiros. Vimos na seção anterior que existem muitos tipos e variações de filas e implementar todas tornaria este capítulo muito extenso e enfadonho. Por isso, apenas algumas implementações serão

realizadas, que são as mais comuns e relevantes. Mais uma vez usaremos os caracteres especiais, tais como `#`, `&`, `@`, entre outros, para exemplificar nossas implementações.

Clássica com ponteiros

Para viabilizar tudo o que foi dito anteriormente em relação às operações, é necessário primeiro realizar dois passos:

1. Criar o registro (`struct`) que representa o valor e contém o ponteiro para os outros elementos da fila, e criar a fila em si, que será um conjunto de registros;
2. Iniciar a fila.

Os códigos são, respectivamente:

```
typedef struct elemento {
 char valor;
 struct elemento *proximo;
} Elemento;

typedef struct fila {
 Elemento *cabeca;
 Elemento *fim;
 int tamanho;
} Fila;

Fila* iniciar() {

 Fila *f = malloc(sizeof(Fila));
 f->cabeca = NULL;
 f->fim = NULL;
 f->tamanho = 0;

 return f;
}
```

Nos dois primeiros códigos, o *struct* chamado `Elemento` tem como finalidade aglutinar o dado que a fila armazenará, assim como o

ponteiro que será usado para conectar os elementos da fila. Já o *struct* chamado `Fila` é responsável por definir a nossa fila de fato, que é um encadeamento de vários *structs* do tipo `Elemento`. Além disso, esse *struct* também armazena o tamanho atual da fila.

No terceiro trecho de código, a função `iniciar()` é responsável por alocar o espaço de memória para a fila e iniciar sua cabeça e fim com `null`, pois nenhum elemento ainda foi enfileirado. É válido ressaltar que, por a fila poder ser manipulada nas suas duas extremidades, temos dois ponteiros, o que é algo novo em relação à pilha, que só tinha um ponteiro, pois somente realizávamos manipulações em uma extremidade (seu *topo*).

A existência desses dois ponteiros facilita muito a manipulação dessa ED. Além disso, esse código também configura o tamanho atual da fila com `0`, pois ela acabou de ser criada. Ao final da configuração, a fila é retornada e está pronta para começar a ser utilizada. Assim, vamos explorar as operações.

ENQUEUE

```
void enqueue(Fila *f, char caractere) { /*I*/

 Elemento *e = malloc(sizeof(Elemento)); /*II*/
 e->valor = caractere;
 e->proximo = NULL;

 if (f->fim != NULL) { /*III*/
  f->fim->proximo = e;
 } else {
  f->cabeca = e;
 }

 f->fim = e; /*IV*/

 f->tamanho = f->tamanho + 1; /*V*/
}
```

O código anterior, responsável por realizar a operação `enqueue`, executa os seguintes passos:

I. Recebe como parâmetro a fila na qual desejamos enfileirar (`f`) um novo elemento e o valor a ser enfileirado (`caractere`);

II. Aloca dinamicamente, com o auxílio da função `malloc`, um novo espaço de memória para armazenar um *struct* do tipo `Elemento`, que é responsável por guardar o valor a ser enfileirado e o ponteiro para o próximo elemento da fila. Coloca no campo `valor` o caractere a ser enfileirado. Por fim, aponta para `null` o ponteiro `proximo` do novo **Elemento** criado e que deve ser enfileirado, pois, como o enfileiramento é sempre ao final, este será o último da fila;

III. Verifica se a fila não está vazia em `if (f->fim != NULL)`. Caso não esteja, executa `f->fim->proximo = e;`, que de fato faz o enfileiramento: aponta o ponteiro `proximo` do *fim da fila* para o novo elemento, o qual já aponta para `null`, indicando que este agora é o último elemento da fila. Se estiver vazia, executa `f->cabeca = e;`, que também de fato faz o enfileiramento, mas agora de forma mais simples, afinal é o primeiro elemento e não temos muito a fazer;

IV. Aponta o *fim da fila* para o novo elemento. Este código tem como finalidade facilitar a manipulação da fila no tocante a enfileiramentos;

V. Incrementa o tamanho da fila em 1.

Podemos então pensar da seguinte maneira: a cada vez que um novo elemento for enfileirado, a fila vai crescendo a partir de seu *fim*. A imagem a seguir ilustra isso. Vamos imaginar que todos os itens da listagem de I até V foram executados duas vezes: a primeira vez para o caractere `$` e a segunda, para o caractere `&`. Temos observações importantes:

- Quando a fila é vazia, *cabeça* e *fim* apontam para o mesmo elemento;
- Quando a fila não é vazia, ela vai "crescendo" para a direita.

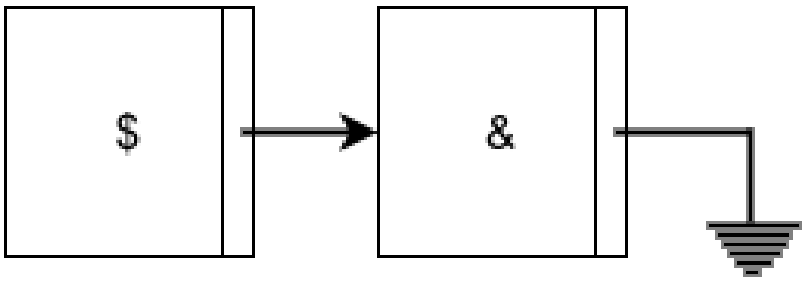


Figura 5.15: Fila após dois enqueues.

SIZE

```
int size(Fila *f) {
   return f->tamanho; /*I*/
}
```

O código anterior é simples e responsável por realizar a operação `size`, que executa o seguinte passo:

I. Retorna o tamanho da fila `f`, que é armazenado no campo `tamanho` do *struct* `Fila`.

Levando em consideração os dois `enqueues` que foram realizados, a execução desse código exibe o valor `2`.

HEAD

```
char head(Fila *f) {
   return f->cabeca->valor; /*I*/
}
```

O código anterior é responsável por executar a operação `head` e realiza a seguinte ação:

I. Retorna o valor da cabeça da fila `f`, que está sendo apontada pelo ponteiro que armazena a cabeça da fila.

No caso, o caractere `$` é exibido, pois foi o primeiro a ser enfileirado.

DEQUEUE

```
char dequeue(Fila *f) {

 Elemento *e;
 char caracter;

 if (!empty(f)) { /*I*/

  e = f->cabeca; /*II*/
  caracter = e->valor;

  f->cabeca = e->proximo; /*III*/
  if (empty(f)) { /*IV*/
   f->fim = NULL;
  }

  f->tamanho = f->tamanho - 1; /*V*/
  free(e);

  return caracter; /*VI*/
 } else {
   printf("Fila vazia.");
   return '\0';
 }
}
```

Menosprezando a explicação das linhas `Elemento *e;` e `char caracter;` , que apenas definem variáveis, o código anterior é responsável por realizar a operação `dequeue` e executa os seguintes passos:

I. Verifica se a fila é vazia. Se for, exibe a mensagem "Fila vazia." e finaliza a execução. Caso contrário, segue ao passo II;

II. Armazena em `e` a cabeça da fila, no caso, seu primeiro elemento. Armazena em `caractere` o valor do elemento da atual cabeça da fila;

III. Transfere a cabeça da fila para `e->proximo`, que é a antiga cabeça da fila e que será desenfileirada;

IV. Verifica se a fila ficou vazia após a mudança da cabeça. Se sim, aponta o fim da fila também para `null`;

V. Decrementa a quantidade de elementos da fila e libera o espaço de memória ocupado pela antiga cabeça da fila;

VI. Retorna o valor da antiga cabeça.

Por fim, levando novamente em consideração os dois `enqueues` feitos anteriormente, o valor retornado pela execução do `dequeue` em nossa fila é `$`, e agora a nova cabeça da fila contém o valor `&`, que foi o segundo elemento a ser enfileirado. Assim, nossa fila fica da seguinte forma:

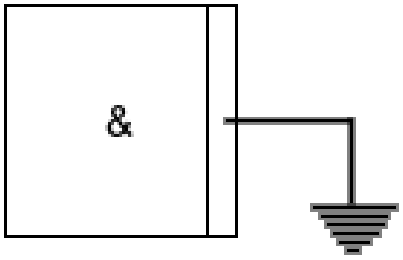


Figura 5.16: Fila após o dequeue.

EMPTY

```
int empty(Fila *f) {
   return f->tamanho == 0; /*I*/
}
```

```
int empty(Fila *f) {
   return f->cabeca == null; /*I*/
}
```

Os códigos anteriores podem ser responsáveis por executar a operação `empty`. Ambos são válidos e surtem o mesmo efeito: informam se a fila está vazia ou não. Levando em consideração o primeiro código, temos o seguinte passo:

I. Retorna o tamanho da fila `f` e verifica se é ele igual a `0`. Se sim, a fila está vazia e o valor `1` é retornado. Caso contrário, o valor `0` é retornado.

Levando em consideração o segundo código, temos o seguinte passo:

I. Verifica se a cabeça da fila `f` é nula (`null`). Se sim, a fila está vazia e o valor `1` é retornado. Caso contrário, o valor `0` é retornado.

Levando em consideração os dois `enqueues` que foram realizados e o `dequeue` executado, a execução desse código exibe o valor `0`. Ou seja, a fila não está vazia, pois ainda possui um elemento.

> Para obter o exemplo completo de nossa fila clássica em C, onde se pode executar (interagindo via prompt) e ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-FilaClassica.git. Lá você conseguirá fazer o download do código e executá-lo.

Circular com vetor

Fila circular com vetor é um dos algoritmos mais interessantes das ED, pois existem controles que ficam navegando através dos índices do vetor para indicar a cabeça e o fim da fila. Aqui, faremos uma implementação com um vetor de `char` de 8 posições.

Para começar, devemos definir a constante com o tamanho de nosso vetor e as variáveis de controle da nossa fila.

```
#define TAMANHO 8
int fila[TAMANHO];

int quantidade, cabeca, fim;
void iniciar() {
    quantidade = 0;
    cabeca = 0;
    fim = 0;
}
```

Nas duas primeiras linhas, temos as definições de nossas variáveis que auxiliaram no manuseio de nossa fila.

No código da função `iniciar`, algumas inicializações são feitas. No caso, a quantidade atual de elementos da fila é iniciada com `0`, sua cabeça e fim também são iniciados com `0`; além disso, o vetor que representará a fila é iniciado com as 8 posições (constante `TAMANHO`). É válido ressaltar que `quantidade`, `TAMANHO`, `cabeca`, `fim` e `fila` são variáveis globais. Após isso, nossa *fila circular* está pronta para funcionar.

ENQUEUE

```
void enqueue(char caractere) {

 if (!full()) { /*I*/

  fila[fim] = caractere; /*II*/
  fim++;
  quantidade++;

  if (fim == TAMANHO) { /*III*/
   fim = 0;
  }
 } else {
  printf("Fila cheia.");
 }
}
```

O código anterior, responsável por realizar a operação `enqueue`, executa os seguintes passos:

I. Verifica se a fila já está cheia (`full`). Se não, permite enfileirar. Isso é necessário, pois vetores têm capacidade fixa e limitada;

II. Coloca no fim da fila o novo elemento, incrementa em 1 o controle do fim da fila, para indicar a nova posição do vetor que representa o fim da fila, e incrementa em 1 a quantidade de elementos da fila;

III. Verifica se, após o enfileiramento, a fila ficou cheia. Se sim, o novo fim volta à cabeça. É justamente esse comportamento que a torna circular.

Levando em consideração que a nossa fila estava vazia e executamos `enqueue('@')` e `enqueue('*')`, temos o resultado na imagem a seguir:

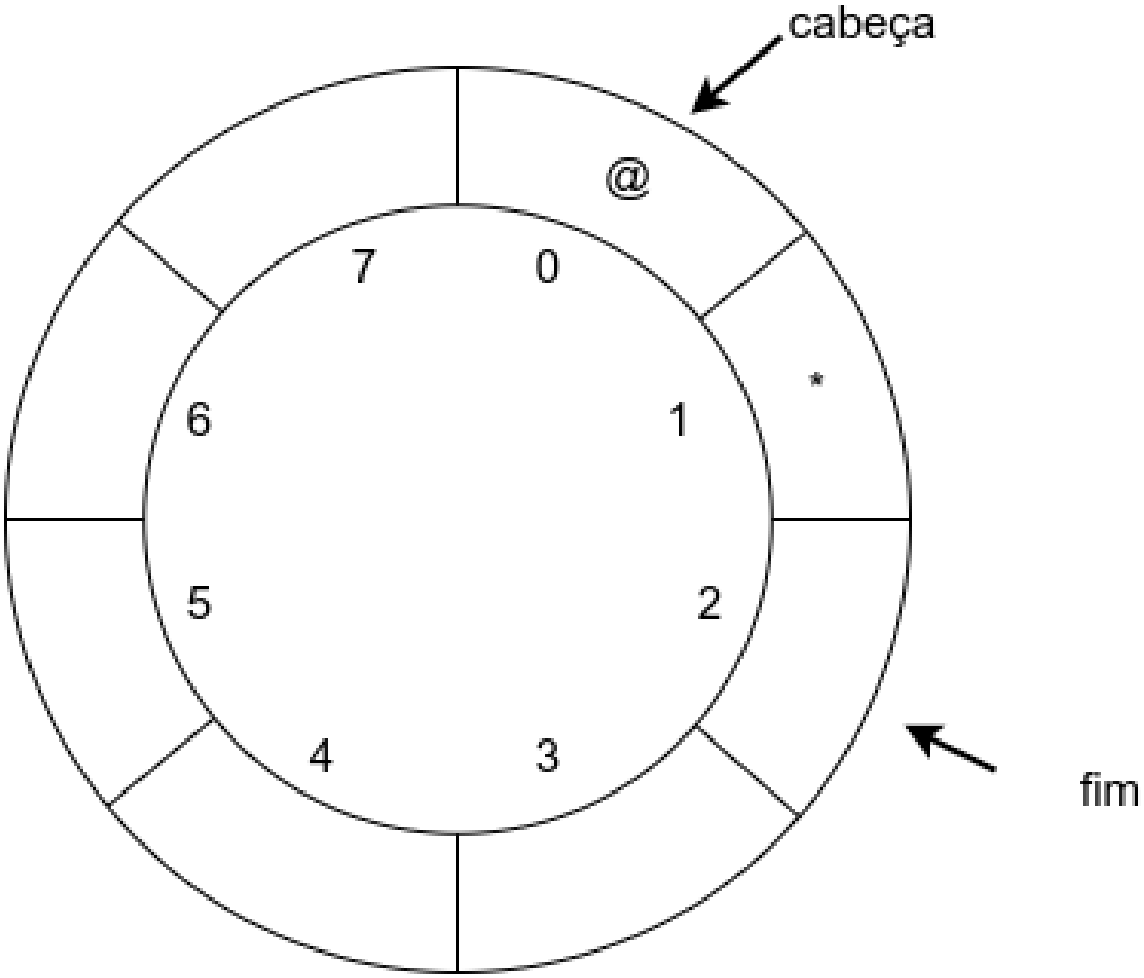


Figura 5.17: Fila circular após enqueue.

SIZE

```
int size() {
   return quantidade; /*I*/
}
```

O código anterior é simples e responsável por realizar a operação `size`, que executa o seguinte passo:

I. Retorna o tamanho da fila, que é armazenado na variável `quantidade`.

Levando em consideração os dois `enqueues` que foram realizados, a execução desse código exibe o valor `2`.

HEAD

```
char head() {
   return fila[cabeca]; /*I*/
}
```

O código anterior é responsável por executar a operação `head` e realiza a seguinte ação:

I. Retorna o valor da cabeça da fila, o qual está armazenado no índice denotado pela variável `cabeca` .

No caso, o caractere `@` é exibido, pois foi o primeiro a ser enfileirado.

DEQUEUE

```
char dequeue() {

 char caractere;

 if (!empty()) { /*I*/

  caractere = fila[cabeca]; /*II*/
  fila[cabeca] = '\0';
  cabeca++;
  quantidade--;

  if (cabeca == TAMANHO){ /*III*/
   cabeca = 0;
  }

  return caractere; /*IV*/
 } else {
  printf("Fila vazia.");
 }
}
```

Ignorando a linha `char caractere;` , que apenas define uma variável, o código anterior é responsável por realizar a operação `dequeue` e

executa os seguintes passos:

I. Verifica se a fila é vazia. Se for, exibe a mensagem "Fila vazia." e finaliza a execução. Caso contrário, segue ao passo II;

II. Armazena em `caractere` o valor atual da cabeça da fila; atualiza a cabeça atual da fila para vazio (`\0`), pois esta será desenfileirada; incrementa o controle `cabeca` para o próximo índice do vetor, para indicar a nova cabeça da fila; e, por fim, decrementa a quantidade de elementos da fila;

III. Verifica se após a mudança da cabeça da fila ela atingiu o índice que representa a capacidade máxima da fila. Se sim, `cabeca` deve receber `0` , para realizar a *circularidade*;

IV. Retorna o valor da antiga cabeça da fila, que foi desenfileirado.

Por fim, levando novamente em consideração os dois `enqueues` feitos anteriormente, o valor retornado pela execução do `dequeue` em nossa fila é `@` , e agora a nova cabeça da fila contém o valor `*` , que foi o segundo elemento a ser enfileirado. Assim, nossa fila fica da seguinte forma:

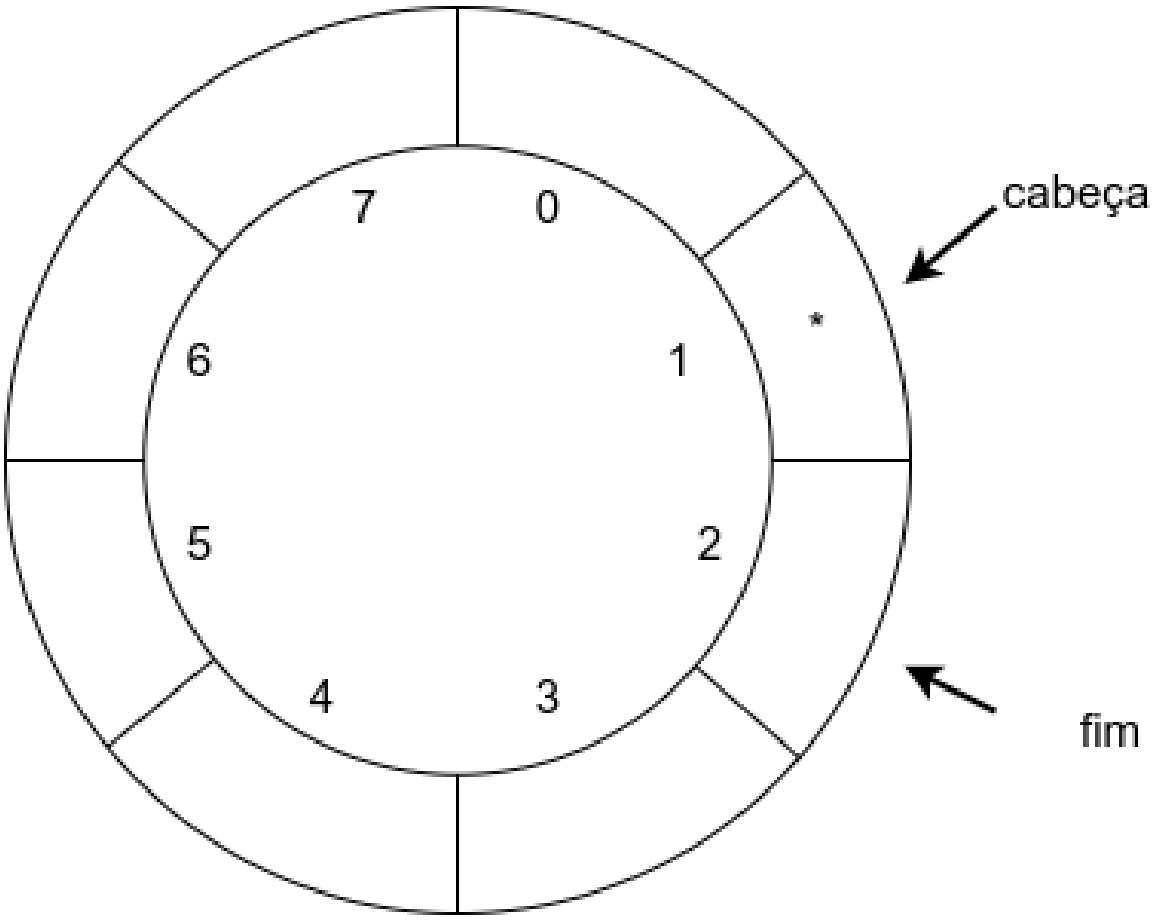


Figura 5.18: Fila circular após o dequeue.

EMPTY

```
int empty() {
 return quantidade == 0;
}
```

O código anterior é responsável por executar a operação `empty`. Ela verifica se a quantidade de elementos atual da fila é `0`. Se sim, `1` é retornado para indicar que está realmente vazia. Caso contrário, `0` é retornado para indicar que não está vazia.

Levando em consideração os dois `enqueues` que foram realizados e o `dequeue` executado, a execução desse código exibe o valor `0`. Ou seja, a fila não está vazia, pois ainda possui um elemento.

FULL

```
int full() {
 return quantidade == TAMANHO;
}
```

A operação `full` só faz sentido quando EDs são criadas com vetores, pois neles temos uma capacidade fixa e limitada de elementos. No caso de nossa *fila circular*, avaliamos se a quantidade de elementos atualmente armazenados na fila é igual à sua capacidade máxima, denotada pela constante `TAMANHO`.

Levando em consideração a primeira vez que usamos essa operação (`full`) na função `enqueue`, o retorno seria `0`, pois a fila estava vazia. Após os dois `enqueues` e o `dequeue` ela ainda retornaria `0`, pois a quantidade ainda seria diferente de 8. Assim, para ela poder estar completamente cheia, teríamos que ter sete `enqueues` consecutivos.

> Para obter o exemplo completo de nossa fila circular em `C`, onde se pode executar (interagindo via prompt) e ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-FilaCircularVetor.git. Lá você conseguirá fazer o download do código e executá-lo.

Prioridade com ponteiro

Como vimos na seção *Tipos*, a *fila de prioridade* possui quatro variações. Implementar todas neste livro seria um exagero! Assim, apenas um tipo será escolhido: a fila com **enfileiramento modificado ascendente**. Neste caso, apenas a operação `enqueue` é modificada. As demais operações ficam na sua forma padrão e, por isso, não serão reapresentadas. É válido ressaltar que, a depender de qual tipo de fila de prioridade for necessário, apenas as operações de `enqueue` e `dequeue` são modificadas. Então sugiro, como prática, a implementação das outras três variações.

Vamos relembrar como a *fila com enfileiramento modificado ascendente* opera:

- Quando um novo elemento é enfileirado, ele se posiciona na fila de acordo com uma ordenação ascendente de prioridade.

Para possibilitar isso, duas alterações devem ser feitas:

1. Acrescentar um novo campo no *struct* `Elemento`, para controlar a prioridade;
2. Alterar o comportamento da função `enqueue`.

O código a seguir satisfaz o item 1:

```
typedef struct elemento {
 char valor;
 int prioridade;
 struct elemento *proximo;
} Elemento;
```

O código a seguir satisfaz o item `2`:

```
void enqueue(Fila *f, char caractere, int prioridade) {

 Elemento *e = malloc(sizeof(Elemento));
 e->valor = caractere;
 e->prioridade = prioridade; /*I*/
 e->proximo = NULL;

 if (f->cabeca == NULL) { /*II*/

  f->cabeca = e;
  f->fim = e;

  f->tamanho = f->tamanho + 1;
 } else {

  if (e->prioridade > f->fim->prioridade) { /*III*/

    f->fim->proximo = e;
    f->fim = e;

    f->tamanho = f->tamanho + 1;
```

```
    return;
  }

  if (e->prioridade < f->cabeca->prioridade) { /*IV*/

    e->proximo = f->cabeca;
    f->cabeca = e;

    f->tamanho = f->tamanho + 1;

    return;
  }

  Elemento *controle = f->cabeca; /*V*/
  Elemento *anterior;

  while (controle->proximo != NULL) { /*VI*/

   if (e->prioridade < controle->prioridade) { /*VII*/

    e->proximo = controle;
    anterior->proximo = e;

    f->tamanho = f->tamanho + 1;

    return;
   } else { /*VIII*/
    anterior = controle;
    controle = controle->proximo;
   }
  }

  free(anterior);
  free(controle);
  anterior = NULL;
  controle = NULL;
 }
}
```

O código anterior tem muitos pontos-chaves, que estão enumerados de I até VIII. Eles são explicados a seguir.

I. A prioridade do novo elemento é armazenada no `Elemento` que acabou de ser criado com a função `malloc` ;

II. Se a cabeça da fila é `null` , então ela está vazia. Assim, a cabeça e o fim recebem no novo elemento (`f->cabeca = e;` e `f->fim = e;`) e o tamanho atual da fila deve ser atualizado (`f->tamanho = f->tamanho + 1`);

III. Se não, verifica se o novo elemento pode ser inserido diretamente ao fim da fila. Se sim, aponta o `proximo` do fim da fila para o novo elemento (`f->fim->proximo = e;`), atualiza o ponteiro do fim da fila (`f->fim = e;`) e incrementa o tamanho (`f->tamanho = f->tamanho + 1;`);

IV. Se não, verifica se o novo elemento pode ser inserido na cabeça da fila. Se sim, aponta o `proximo` do novo elemento para a cabeça da fila (`e->proximo = f->cabeca;`), atualiza o ponteiro da cabeça da fila (`f->cabeca = e;`) e incrementa o tamanho (`f->tamanho = f->tamanho + 1;`);

V. Se a execução chegar a esse passo, é sinal de que é necessário navegar pela fila para encontrar o local do novo elemento. Para isso, cria-se um `controle` para navegar na fila a partir de sua cabeça e um `anterior` para não "desconectar" a fila;

VI. Enquanto o `controle` não chegar ao fim da fila, no caso `NULL` , fazemos:

VII. Se a prioridade do novo elemento (`e->prioridade`) for menor do que a do elemento corrente (`controle->prioridade`), o `proximo` do novo elemento recebe o `controle` (`e->proximo = controle;`), que é o elemento que deve ficar à frente do novo elemento, e o `proximo` do elemento anterior ao novo elemento é conectado a este (`anterior->proximo = e;`). Com isso, mantemos a fila íntegra;

VIII. Se a prioridade do novo elemento (`e->prioridade`) for maior, então é necessário ficar navegando na fila até que o item `VII` seja verdade. Para isso, o `anterior` deve navegar para o atual `controle` (`anterior = controle;`) e o `controle` deve seguir adiante (`controle = controle->proximo;`).

Vale ressaltar que os ponteiros auxiliares `controle` e `anterior` devem ser desalocados com a função `free` ao fim da função.

> Para obter o exemplo completo de nossa fila de prioridade em C, onde se pode executar (interagindo via prompt) e ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-FilaPrioridade.git. Lá você conseguirá fazer o download do código e executá-lo.

Com dois finais (Dupla, Deque, Dequeue, Com duas extremidades) com vetor

Devido a esse tipo de fila possibilitar duas novas operações — *enfileirar na cabeça* e *desenfileirar no fim* — além das já existentes na *fila clássica*, somente essas novas operações serão exploradas. As outras duas seguem os comportamentos já explicados. Vale lembrar que esse tipo de fila deve ser *circular* e, preferencialmente, com vetor de capacidade par. Sendo assim, as implementações de *fila circular com vetor* podem ser reaproveitadas aqui. Outra pequena diferença é que a cabeça e o fim devem ser iniciados com `-1` para facilitar sua manipulação.

Assim, vejamos a seguir o código para *enfileirar na cabeça* (`enqueue_head`), *desenfileirar no fim* (`dequeue_fim`) e a modificação da inicialização da fila (`iniciar`).

```
void iniciar() {

 quantidade = 0;
```

```
 cabeca, fim = -1;
}

void enqueue_head(char caractere) {

 if (full()) { /*I*/
  printf("Fila cheia!");
 } else {

    if (empty()) {  /*II*/

     cabeca = 0;
     fim = 0;
    } else {

     if (cabeca == 0) {  /*III*/
      cabeca = TAMANHO - 1;
     } else {
      cabeca = cabeca - 1;
     }
    }

  quantidade = quantidade + 1;
  fila[cabeca] = caractere;
 }
}
```

Os pontos-chaves do código anterior são explicados a seguir:

I. Se a fila estiver cheia, nada mais pode ser enfileirado;

II. Se não estiver cheia, verifica se está vazia, ou seja, se é o primeiro enfileiramento. Se for, a fila tem sua cabeça e fim iniciados para `0`. Posteriormente, a quantidade é incrementada em 1 (`quantidade = quantidade + 1;`) e a cabeça recebe o elemento a ser enfileirado (`fila[cabeca] = caractere;`);

III. Se não estiver vazia, então temos que ver se a cabeça ainda se encontra na posição `0`. Se sim, o novo elemento vai para a nova cabeça (posição `TAMANHO - 1`). Se não, vai para a posição `cabeca =`

`cabeca - 1` . Isso é necessário para mandar a circularidade e a característica de "dois finais". Posteriormente, a quantidade é incrementada em 1 (`quantidade = quantidade + 1;`), e a cabeça recebe o elemento a ser enfileirado (`fila[cabeca] = caractere;`).

```
char dequeue_fim() {

 char caractere;

 if (empty()) { /*I*/

    printf("Fila vazia!");
    return '\0';
 } else {

  caractere = fila[fim]; /*II*/
  fila[fim] = '\0';

  if (cabeca == fim) { /*III*/

   cabeca = -1;
   fim = -1;
  } else {

   if (fim == 0) { /*IV*/
    fim = TAMANHO - 1;
   } else {
    fim = fim + 1;
   }
  }

  quantidade = quantidade - 1;
  return caractere;
 }
}
```

Novamente, o código anterior possui pontos-chaves, que são explicados a seguir:

I. Se a fila estiver vazia, nada mais pode ser desenfileirado;

II. Se não estiver vazia, então armazena-se o último elemento da fila e limpa-o da fila (`caractere = fila[fim];` e `fila[fim] = '\0';`);

III. Se o passo anterior fizer a cabeça ficar igual ao fim (`cabeca == fim`), então esvaziamos a fila completamente. Assim, `cabeca = -1;` e `fim = -1;` . Posteriormente, a quantidade de elementos é decrementada em 1 e o elemento é retornado (`quantidade = quantidade - 1;` e `return caractere;`);

IV. Se não era o último elemento da fila, então verifica se o fim está na posição `0` . Se sim, este vai para `fim = TAMANHO - 1` . Se não, vai para `fim = fim + 1` . Posteriormente, a quantidade de elementos é decrementada em 1 e o elemento é retornado (`quantidade = quantidade - 1;` e `return caractere;`).

Esse tipo de fila é a mais difícil de visualizar mentalmente e entender. Aconselho fortemente um processo de "*debug*" com papel e caneta para entender seu funcionamento. Faça o passo a passo dos dois códigos anteriormente apresentados.

> Para obter o exemplo completo de nossa fila com dois finais em C, onde se pode executar (interagindo via prompt) e ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-FilaDeque.git. Lá você conseguirá fazer o download do código e executá-lo.

5.5 Exemplos de uso

Assim como a *pilha*, a *fila* também possui uma grande gama de aplicabilidades dentro da computação e no nosso dia a dia. A seguir, listamos alguns exemplos.

Fila de impressão

Esse talvez seja o exemplo mais clássico do uso de filas. Toda vez que alguém está em seu local de trabalho e necessita que algum documento seja impresso, ele é enviado à impressora. Mas, imaginemos uma empresa ou setor com 100 pessoas. Se todas mandassem, coincidentemente, uma impressão ao mesmo tempo, quem deveria ter sua impressão realizada primeiro? Difícil determinar, não é? Nesse caso, uma fila entra em ação.

Para a *fila de impressão*, não importa se é um simples funcionário ou o dono da empresa. Quem chega primeiro, é enfileirado e sairá primeiro (filosofia FIFO). Dessa forma, se consegue ser "imparcial" e evitar que impressões fiquem "retidas" na fila devido a não se ter uma ordem de execução.

Execução de tarefas no SO

Sistemas operacionais (SO) são responsáveis por utilizar e gerenciar recursos como memória RAM e CPU. Quando um SO está em execução, ele fica constantemente bloqueando e liberando recursos de acordo com os processos em execução. E para conseguir realizar essa tarefa, mais uma vez filas podem ser utilizadas.

Nesse caso, a fila auxilia o SO a não consumir de forma "gulosa" recursos como memória e processador, o que causa o travamento do sistema. Dessa forma, cada processo que necessita de determinadas quantidades de memória e CPU vai sendo enfileirado e executado segundo a política FIFO. Assim, cada processo espera os recursos serem liberados para começar a executar, e quando finalizarem suas execuções, liberam esses recursos para os próximos da fila.

Sempre que cada processo finaliza sua execução, ele é desenfileirado e libera seus recursos anteriormente alocados. Dessa forma, o próximo processo pode receber os recursos necessários e iniciar sua execução.

O algoritmo mais famoso de escalonamento (execução) de processos em SOs e que usa o conceito de fila é o *Round-Robin*, e ele usa uma fila circular.

Fila de um hospital com prioridade

Sabemos que hospitais são locais onde as coisas devem acontecer de forma rápida, para evitar situações que causem danos à saúde das pessoas. Por isso, vemos que às vezes pessoas que chegaram muito depois de outras são atendidas primeiro, mesmo estando na mesma fila de atendimento. No caso, isso acontece devido à *prioridade* do atendimento dessa pessoa ser maior do que a das demais na fila.

Assim, quando um novo paciente entrar na fila de atendimento, ele deve ter uma *prioridade*, determinada de acordo com seu estado de saúde. O posicionamento na fila será feito a partir da prioridade determinada, a depender da situação do paciente. Isso pode fazer com que essa pessoa "pule" posições da fila para logo ser atendida, devido à sua enfermidade.

Fila para sistemas distribuídos

É comum sistemas distribuídos necessitarem que determinados processos sejam executados de forma mais rápida, por possuírem uma prioridade maior em um determinado momento, e que outros possam esperar mais por sua execução.

Nesses casos, uma *fila dupla* pode ser utilizada para escalonar a execução de processos e, ao mesmo tempo, represar outros processos. Isso é possibilitado pela natureza da *fila dupla*: enfileirar e desenfileirar em ambas as extremidades e, também, por se comportar como se duas filas existissem e fossem "independentes", como visto na seção anterior. Dessa forma, os processos vão sendo escalonados sob demanda e não a partir de prioridades.

5.6 Exercícios

> Todos os exercícios devem usar filas com ponteiros.

1) Levando em consideração uma fila inicialmente vazia e que as seguintes operações foram realizadas, responda:

1. `dequeue()`
2. `enqueue(f, b)`
3. `dequeue()`
4. `head()`
5. `size()`
6. `empty()`
7. `enqueue(f, c)`
8. `dequeue()`
9. `enqueue(f, d)`
10. `head()`
11. `dequeue()`
12. `dequeue()`
13. `empty()`

a) Qual o retorno do passo 1?

b) Qual o retorno do passo 4?

c) Qual o retorno do passo 5?

d) Qual o retorno do passo 10?

e) Qual o retorno do passo 13?

2) Implemente um programa em C que manipule 3 filas de números inteiros. A primeira, chamada de `f`, deve conter os números de 1 até 15. A segunda, chamada de `f_par`, está inicialmente vazia, assim como a terceira, chamada de `f_impar`. O programa deve

transferir os números pares de `f` para a fila `f_par` e os números ímpares de `f` para a fila `f_impar`.

3) Implemente um programa em C que manipule 2 filas de `char`. A primeira, chamada de `f1`, deve conter os caracteres `A`, `C`, `E`, `G`, `I`, `K`, `M`, `O`, `Q`, `S`, `U`, `W` e `Y`. A segunda, chamada de `f2`, deve conter os caracteres `B`, `D`, `F`, `H`, `J`, `L`, `N`, `P`, `R`, `T`, `V`, `X` e `Z`. O programa deve exibir o alfabeto na ordem de `A` até `Z`.

4) Escreva uma função em C que recebe duas filas, `f1` e `f2`, ambas contendo números inteiros. Os elementos dessas filas apresentam-se em ordem crescente, ou seja, o menor valor de cada fila encontra-se no início dela. A partir delas, crie uma terceira fila de números inteiros também em ordem crescente formada pelos elementos que se encontram nas filas `f1` e `f2`. Por exemplo, se `f1 = {2, 10, 15, 25}`, onde `2` é o início da fila, e `f2 = {3, 5, 9, 15}`, onde `3` é o início, a fila `f3` construída deve ser `{2, 3, 5, 9, 10, 15, 15, 25}`, onde `2` é o início e `25` é o fim da fila.

5) Levando em consideração os conceitos estudados nas seções anteriores, como a fila se encontraria após a execução do código a seguir?

```
int main(void) {

  Fila *f;
  int i;
  inicialize(&f);

  for(i=0;i<=12;i=i+2){

   if(i % 6 == 0){
    dequeue(&f);
   }else{
    enqueue(&f,3*i)
   }
  }
```

```
  return 0;
}
```

Leve em consideração que a fila foi inicializada com sucesso e que, nesse caso, se encontra vazia no início da execução desse código.

Capítulo 6
Lista

Neste capítulo, estudaremos a estrutura de dados **lista** (do inglês *list*). Conceitos-chaves e definições serão abordadas, assim como aplicações na computação e no nosso dia a dia. O que já aprendemos acerca do uso de ponteiros em capítulos anteriores também se aplica a este capítulo de *lista* e, por isso, não precisa ser explicado de forma detalhada.

6.1 Fundamentos

Podemos definir uma *lista* como:

> Uma estrutura de dados onde os elementos são linearmente organizados.

A partir dessa definição, podemos perceber que essa estrutura de dados é mais simples no tocante a seu funcionamento. Ao contrário da *pilha* e da *fila*, sua definição não tem nada que acarrete um comportamento mais rebuscado. **Ninguém está em cima de ninguém**; **ninguém está ordenado e sequenciado**: simplesmente os elementos estão ligados de forma linear.

Da mesma forma que pilhas e filas, as *listas* também aparecem em nosso dia a dia — por exemplo: lista de compras; lista de chamada de uma disciplina de graduação; manual (lista de passos) para montar um armário etc. Ou seja, essa ED é visivelmente muito comum e, mais uma vez, nos computadores não é diferente.

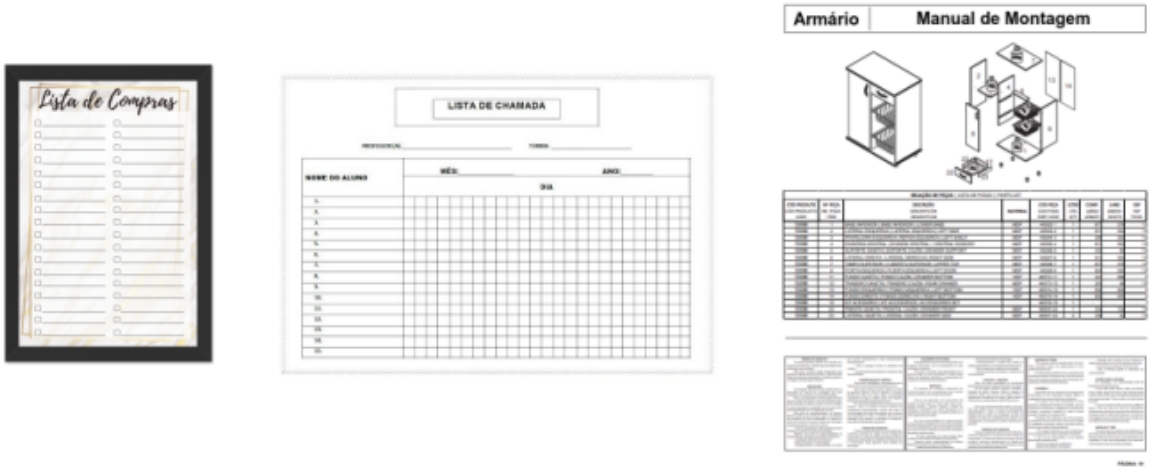


Figura 6.1: Listas.

Diferente de pilhas e filas, a lista tem um "comportamento livre" relativo a seu uso, ou seja, não temos algo similar a FILO ou LIFO. Basta os elementos estarem linearmente organizados que já teremos uma lista. Novamente, na memória do computador, os elementos são ligados através de seus endereços de memória, quando implementados com ponteiros.

Para elucidar o que de fato é uma lista, podemos explorar o exemplo citado da "lista de compras". Sabemos que essa lista possui uma *quantidade inicial* de itens, mas que pode *aumentar*, caso seja notado durante a compra que itens estavam faltando. Geralmente, mas não obrigatoriamente, esses novos itens vão sendo acrescentados ao final da lista. Também sabemos que podemos *eliminar* itens de forma não sequencial, pois vamos "riscando" da lista os que vão sendo comprados de acordo com o momento em que vamos nos deparando com eles no caminhar pelo supermercado. A eliminação não tem uma sequência definida. Podemos também *alterar* a lista. Por exemplo, se inicialmente tínhamos colocado uma determinada quantidade de um item, podemos comprar mais ou menos dele. Ou seja, modificamos pontualmente dentro da lista um de seus elementos.

6.2 Operações

Por a ED *lista* não ter um comportamento bem definido ou uma política de uso como a *pilha* e a *fila*, existe uma facilidade maior de manipulação, o que leva a uma maior quantidade de operações disponíveis. Mas é válido ressaltar que essa quantidade maior de operações não implica que suas implementações sejam mais complexas. As operações disponíveis para listas são as a seguir:

- `ADD` : adiciona um novo elemento, sempre no começo ou no fim da lista;
- `EMPTY` : verifica se a lista está vazia;
- `ADD` com posição: adiciona um novo elemento em uma posição específica;
- `GET` : obtém o elemento de uma posição específica;
- `SET` : modifica um elemento de uma posição específica;
- `DELETE` : exclui um elemento de uma posição específica;
- `SIZE` : obtém o tamanho atual da lista.

A seguir, essas operações serão detalhadas e, para isso, levaremos em consideração uma lista fictícia chamada de `L`, que se encontra inicialmente vazia e pronta para uso.

ADD

`ADD` é a operação de adicionar um novo elemento na lista, seja em seu início ou fim. Geralmente, a adição no início é mais utilizada por ser mais simples. Assim, considerando a nossa lista `L` e o elemento `&`, para que ele seja adicionado, temos: `add(L,'&');`. O resultado dessa execução está a seguir:

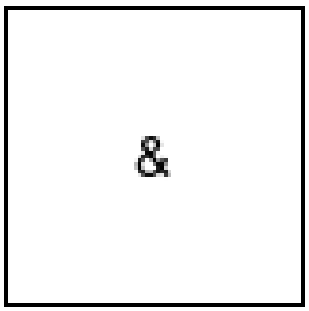


Figura 6.2: Lista após add.

ADD com posição

Assim como o `add` inicial, essa versão com posição tem a capacidade de adicionar um novo elemento na lista. Porém, nessa variação da operação `add` é possível indicar em qual posição o novo elemento deve ser adicionado. Podemos agora adicionar um elemento no início de nossa lista `L`, mais especificamente na posição `1`. Dessa maneira, podemos ter `add_pos(L, '@', 1);`, e a lista fica agora com dois elementos: `@` e `&`, nas posições `1` e `2`, respectivamente. Aditivamente, faremos um outro `add_pos`, mas agora na posição `2`, no caso `add_pos(L, '#', 2);`. Dessa forma, deslocamos o elemento `&` para a posição `3`. O resultado dessas execuções está a seguir:

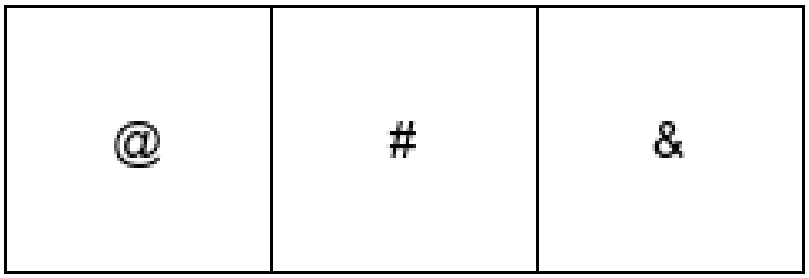


Figura 6.3: Lista após add na posição 1 e 2.

EMPTY

EMPTY é a operação que indica se a lista está vazia, ou seja, se seu tamanho atual é `0` ou se ela ainda se encontra nula (`null`). Novamente, levando em consideração nossa lista `L`, temos:

`empty(L)`. O resultado dessa execução é `false`, pois na seção anterior os elementos `&`, `@` e `#` foram adicionados.

GET

GET é a operação que retorna o valor de uma determinada posição da lista. Inicialmente, pode se pensar que ela é similar ao `pop` ou `dequeue`. Entretanto, ao contrário destas, não há retirada do elemento da estrutura, mas apenas o retorno de seu valor. Posto isso, se as operações `get(2)`, `get(1)` e `get(3)` forem executadas na nossa lista `L`, teremos os respectivos resultados: `#`, `@` e `&`.

SET

SET é uma operação que modifica o valor de um elemento em uma determinada posição na lista. Dessa maneira, para alterarmos o valor do elemento da primeira posição de nossa lista, temos a seguinte operação: `set(L, '!', 1);`. Após a execução dessa operação, o valor `@` é alterado por `!`. A lista, então, fica assim:

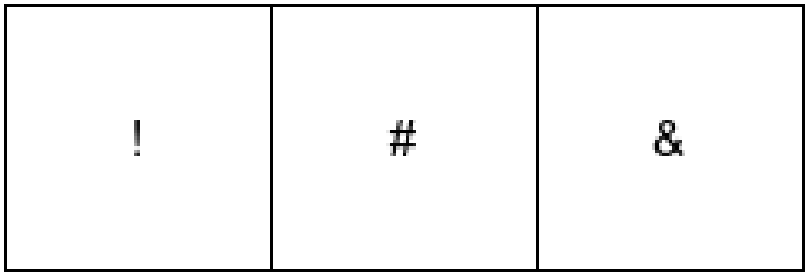


Figura 6.4: Lista após set na posição 1.

DELETE

DELETE é a operação que exclui um elemento de uma determinada posição da lista. Este, sim, se comporta como o `pop` e `dequeue`. A retirada da estrutura retorna o valor do elemento eliminado. Então, se na nossa lista `L` executarmos a operação `remove(L, 2);`,

excluiremos o elemento na posição `2`, no caso o `#`. Após essa execução, a lista se encontra na situação da imagem a seguir:

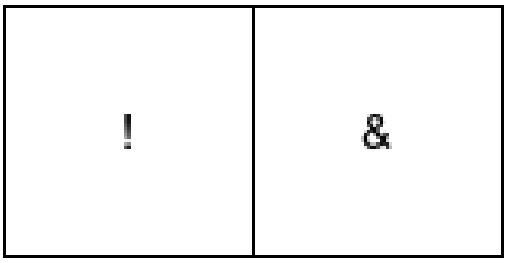


Figura 6.5: Lista após remove na posição 2.

SIZE

SIZE, por fim, é a operação que indica o tamanho atual da lista. Esse valor pode aumentar ou diminuir à medida que elementos são adicionados ou excluídos da lista. Levando em consideração nossa lista `L` de exemplo, temos o resultado `2` como valor atual da execução da operação `size(L)`.

6.3 Tipos de listas

Assim como a *fila*, a *lista* possui mais de um tipo, que vamos ver nas subseções a seguir.

Simplesmente encadeada

A *lista simplesmente encadeada* é a mais comumente usada. Ela é basicamente a execução das operações anteriormente apresentadas e apenas com um único sentido de navegação: início → fim. Ou seja, se tivermos uma lista com 10 posições e for executada a operação `get(7)`, teremos que percorrer a partir do início da lista até

a posição `7`. As operações podem ser usadas na ordem necessária e de acordo com a demanda.

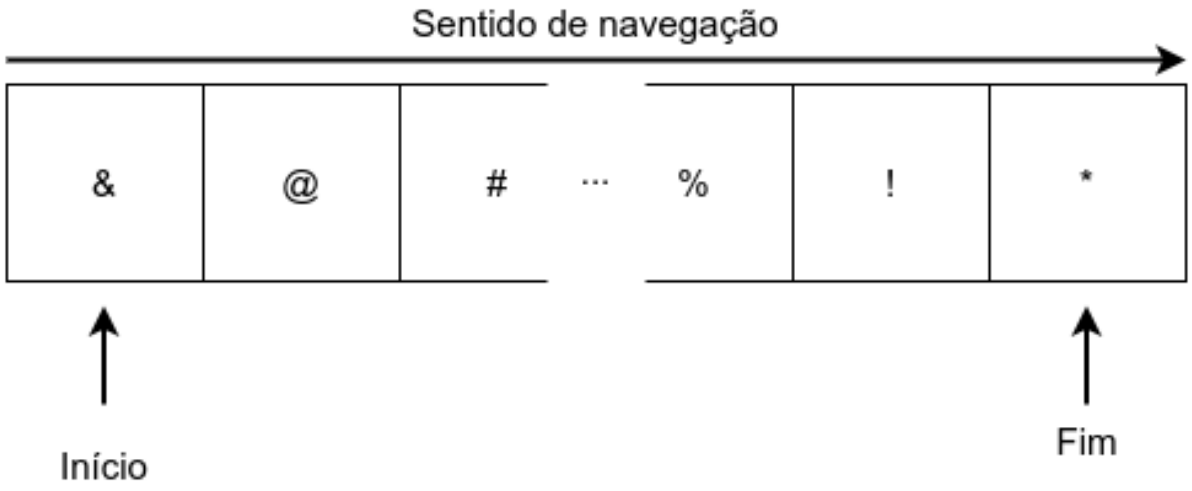


Figura 6.6: Lista simplesmente encadeada.

Duplamente encadeada

A *lista duplamente encadeada* é uma lista que possui as operações idênticas à *lista simplesmente encadeada*. Contudo, a diferença está em seu funcionamento interno, em seu sentido de navegação.

Nas listas duplamente encadeadas, temos dois sentidos de navegação: início → fim (esquerda/direita) e fim → início (direita/esquerda), possibilitando a navegação em ambos os sentidos. Isso é possível porque listas duplas têm a capacidade de ter o conhecimento do seu próximo elemento assim como de seu antecessor. Além disso, elas podem memorizar o elemento atual, que foi acessado pela última operação `get`. Por exemplo, se tivermos uma lista com 15 posições e for executado um `get(13)` e depois um `get(9)`, o segundo `get` não precisará ir ao início da lista até chegar à posição `9`, mas sim executará a partir da posição `13`. Dessa forma, será realizada uma navegação no sentido fim → início, passando pelas posições `12`, `11`, `10` e chegando na `9`, pois a posição `9` antecede a posição `13`.

Embora esse novo sentido de navegação possa parecer preciosismo, essa possibilidade traz consigo um grande ganho de performance, pois diminui a quantidade de itens visitados na lista. Esse ganho se demonstra maior ainda em listas com grandes quantidade de itens (centenas ou milhares).

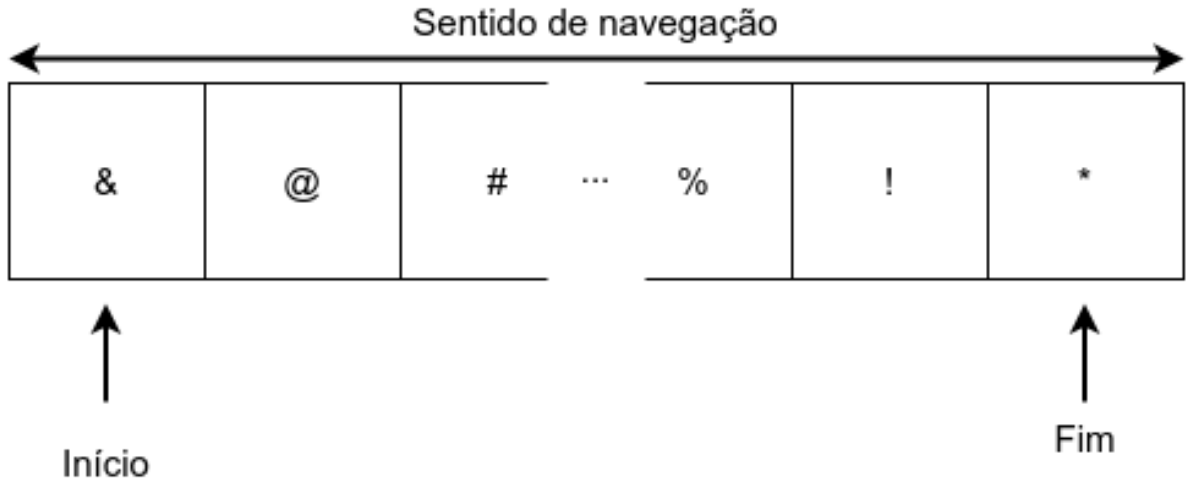


Figura 6.7: Lista duplamente encadeada.

Circular

Esse tipo de lista segue o mesmo princípio de *fila circular*, em que as extremidades se conectam. Entretanto, como a *lista* é uma ED mais livre em relação à sua manipulação, os conceitos de início e fim não fazem muito sentido para esse tipo de lista. Novamente, todas as operações apresentadas também se aplicam a esse tipo: pode ser criada uma lista circular simplesmente ou duplamente encadeada, e os conceitos inerentes a esses tipos também se replicam aqui.

6.4 Implementação

Nesta seção, veremos os códigos em `c` para podermos realizar as operações disponíveis para a ED lista. É válido ressaltar que listas

podem também ser implementadas com vetores. Todavia, essa implementação é muito simples e, no fim, é a manipulação de um vetor como qualquer outro, apenas com a limitação de tamanho fixo.

Neste livro, usamos preferencialmente a *implementação com ponteiro*, que é um pouco mais difícil, mas é a mais utilizada, devido à capacidade de ter seu tamanho aumentado de acordo com a necessidade. Mais uma vez, nossa estrutura será constituída de caracteres especiais, tais como `%`, `!`, `&`, entre outros. Assim como nas EDs anteriores, as implementações serão por tipos, para conhecermos as peculiaridades de cada um.

Simplesmente encadeada

Para iniciar nossas implementações, é necessário primeiro realizar dois passos:

1. Criar o registro (`struct`), que representa o valor e contém o ponteiro para os outros elementos da lista, e criar a lista em si, que será um conjunto de registros;
2. Iniciar a lista.

Os códigos são, respectivamente:

```
typedef struct elemento {
 char valor;
 struct elemento *proximo;
} Elemento;

typedef struct lista {
 Elemento *inicio;
 int tamanho;
} Lista;

Lista* iniciar() {

 Lista *l = malloc(sizeof(Lista));
 l->inicio = NULL;
 l->tamanho = 0;
```

```
 return l;
}
```

Nos dois primeiros códigos, o `struct` chamado **Elemento** tem como finalidade aglutinar o dado que a lista armazenará e o ponteiro que será usado para conectar os elementos da lista. Já o `struct` chamado **Lista** é responsável por definir a nossa lista de fato, que é um encadeamento de vários *structs* do tipo **Elemento**. Além disso, esse *struct* também armazena o tamanho atual da lista e seu início.

O último trecho de código, a função `iniciar()` é responsável por alocar o espaço de memória para o primeiro elemento da lista e iniciá-lo com `NULL`, pois nenhum elemento foi adicionado ainda. Além disso, esse código também configura o tamanho atual da lista com `0`, pois ela acabou de ser criada. Ao final da configuração, a lista é retornada e está pronta para começar a ser utilizada.

Com a lista definida e iniciada, vamos explorar as operações.

ADD

```
void add(Lista *l, char caractere) {

 if (empty(l)) {

  Elemento *e = malloc(sizeof(Elemento)); /*I*/
  e->valor = caractere;
  e->proximo = NULL;

  l->inicio = e;
 } else {

  Elemento *e = malloc(sizeof(Elemento)); /*II*/
  e->valor = caractere;
```

```
  Elemento *antigo_inicio = l->inicio;
  l->inicio = e;
  e->proximo = antigo_inicio;
 }

 l->tamanho = l->tamanho + 1;
}
```

O código anteriormente apresentado é responsável por:

I. Detectar se a lista está vazia e, se sim, criar um novo **Elemento** (`Elemento *e = malloc(sizeof(Elemento));`) para armazenar o valor do novo elemento (`e->valor = caractere;`). Como a lista estava vazia, não temos um próximo elemento (`e->proximo = NULL;`), ou seja, este é o primeiro. Ao final, a quantidade de elementos de lista é incrementada (`l->tamanho = l->tamanho + 1;`).

II. Se a lista não está vazia, o novo elemento é adicionado em seu início. Assim, este novo **Elemento** é criado (`Elemento *e = malloc(sizeof(Elemento));`) para armazenar o novo valor (`e->valor = caractere;`). Para não perder o ponteiro para a lista no momento de adicionar em seu início, é necessário guardar seu antigo início (`Elemento *antigo_inicio = l->inicio;`). Após isso, o novo início é colocado no seu devido lugar (`l->inicio = e;`) e o novo elemento (que é o novo início) é apontado para o antigo início (`e->proximo = antigo_inicio;`). Com isso, não perdemos o encadeamento dos elementos. Por fim, a quantidade de elementos da lista é incrementada (`l->tamanho = l->tamanho + 1;`).

EMPTY

```c
int empty(Lista *l) {
   return l->tamanho == 0; /*I*/
}

int empty(Lista *l) {
   return l->inicio == NULL; /*II*/
}
```

Os códigos anteriores são responsáveis por executar a operação `empty` em nossa lista. Ambos são válidos e surtem o mesmo efeito: informam se a lista está vazia ou não. Levando em consideração o primeiro código, temos o seguinte comportamento:

I. Retorna o tamanho da lista `l` e verifica se é igual a `0`. Se sim, a lista está vazia e o valor `1` é retornado. Caso contrário, o valor `0` é retornado.

Levando em consideração o segundo código, temos:

II. Verifica se a lista `l` não tem nenhum elemento. Se sim, a lista está vazia e o valor `1` é retornado. Caso contrário, o valor `0` é retornado.

Levando em consideração o `add` que foi realizado anteriormente, a execução de qualquer um destes códigos exibirá o valor `0`. Ou seja, a lista não está vazia, pois possui um elemento.

ADD com posição

```
void add_pos(Lista *l, char caractere, int posicao) {

 if (empty(l)) { /*I*/
  printf("Lista vazia. Add posicional não permitido!\n");
  return;
 }

 if (posicao > l->tamanho || posicao <= 0) { /*II*/
  printf("Posição inválida!\n");
  return;
 }

 if (posicao == 1) { /*III*/
  add(l, caractere);
  return;
 }

 Elemento *e_atual = l->inicio; /*IV*/
```

```
 Elemento *e_anterior;

 Elemento *e = malloc(sizeof(Elemento)); /*V*/
 e->valor = caractere;

 int i; /*VI*/
 for (i = 1; i < posicao ; i++) {
  e_anterior = e_atual;
  e_atual = e_atual->proximo;
 }

 e_anterior->proximo = e; /*VII*/
 e->proximo = e_atual;

 l->tamanho = l->tamanho + 1; /*VIII*/
}
```

O código de `add_pos` apresentado é responsável por:

I. Verificar se a lista está vazia. Se sim, não temos como adicionar em nenhuma posição;

II. Se não estiver vazia, é necessário validar a posição;

III. Caso a posição seja válida e seja `1`, então o `add_pos` pode se comportar como o `add` e tal função é executada;

IV. Caso a posição não seja `1`, então temos que detectar o local onde a inserção será realizada. Para isso, inicialmente criamos um ponteiro auxiliar para começar a navegar na lista a partir de seu início (`Elemento *e_atual = l->inicio;`) e também criamos um ponteiro que auxiliará no momento da inserção (`Elemento *e_anterior;`), para manter a lista íntegra;

V. O novo **Elemento** é criado (`Elemento *e = malloc(sizeof(Elemento));`) para armazenar um novo valor (`e->valor = caractere;`) que será inserido na lista;

VI. Neste ponto, é feita a procura pela posição de inserção. Para isso, navegamos até acharmos a posição desejada (`for (i = 1; i < posicao ; i++)`). Enquanto essa navegação é realizada, ficamos movimentando os ponteiros auxiliares `e_anterior` e `e_atual` para que eles se posicionem antes (`e_anterior`) e depois (`e_atual`) da posição de inserção;

VII. Quando a posição é atingida, então o `próximo` do anterior recebe o novo elemento (`e_anterior->proximo = e;`) e o `próximo` do novo elemento recebe o `atual` (`e->proximo = e_atual;`). Com isso, conseguimos manter a lista ligada através de seus ponteiros;

VIII. Por fim, a quantidade de elementos é incrementada em `1` (`l->tamanho = l->tamanho + 1;`).

GET

```
char get(Lista *l, int posicao) {

 if (empty(l)) { /*I*/
  printf("Lista vazia.\n");
  return '\0';
 }

 if (posicao > l->tamanho || posicao <= 0) { /*II*/
  printf("Posição inválida!\n");
  return '\0';
 }

 int i = 1; /*III*/
 Elemento *e = l->inicio;
 while (e->proximo != NULL) {

  if (i == posicao) {
   return e->valor;
  } else {

   e = e->proximo;
   i++;
```

```
  }
 }

 return '\0'; /*IV*/
}
```

O código anterior é responsável por:

I. Verificar se a lista é vazia. Se for, não temos como obter elementos de nenhuma posição;

II. Se não for vazia, precisamos saber se é uma posição válida;

III. Se tudo estiver certo, então a procura pela posição e, consequentemente, do elemento é iniciada. Para isso, vamos para o início da lista (`Elemento *e = l->inicio;`) e navegamos por ela enquanto ela não acabar (`while (e->proximo != NULL)`). Se atingirmos a posição desejada (`if (i == posicao)`), o valor do elemento que corresponde à posição é retornado (`return e->valor`); se não, passamos para o próximo elemento e incrementamos a posição (`e = e->proximo;` e `i++;`);

IV. Se este ponto do código for atingido, é sinal de que algo "deu errado" e a posição não foi encontrada. Assim, vazio (`\0`) é retornado.

É válido ressaltar que, como validações de *vazio* e *posição* são feitas de forma prévia, é de se esperar que o ponto III seja atingido e um elemento seja retornado. Entretanto, quando se trata de C e ponteiros, uma retorno vazio (IV) é fornecido para evitar comportamentos adversos, caso a execução atinja esse ponto de forma inesperada.

SET

```
void set(Lista *l, char caractere, int posicao) {

 if (empty(l)) { /*I*/
  printf("Lista vazia. Set não permitido!\n");
```

```
  return;
 }

 if (posicao > l->tamanho || posicao <= 0) { /*II*/
  printf("Posição inválida!\n");
  return;
 }

 int i = 1; /*III*/
 Elemento *e = l->inicio;
 while (e->proximo != NULL) {

  if (i == posicao) {
   e->valor = caractere;
   return;
  } else {

   e = e->proximo;
   i++;
  }
 }
}
```

O código anterior é responsável por:

I. Verificar se a lista é vazia. Se for, não temos como modificar elementos de nenhuma posição;

II. Se não for vazia, precisamos saber se é uma posição válida;

III. Se tudo estiver certo, então a procura pela posição e o elemento a ser atualizado é iniciada. Para isso, vamos para o início da lista (`Elemento *e = l->inicio;`) e navegamos por ela enquanto ela não acabar (`while (e->proximo != NULL)`). Se atingirmos a posição desejada (`if (i == posicao)`), o valor do elemento que corresponde à posição é alterado (`e->valor = caractere;`). Se não, passamos para o próximo elemento e incrementamos a posição (`e = e->proximo;` e `i++;`).

É válido ressaltar que, como validações de vazio e posição são feitas de forma prévia, se o ponto III é atingido, com certeza um elemento terá seu valor alterado.

DELETE

```
char delete(Lista *l, int posicao) {

 if (empty(l)) { /*I*/
  printf("Lista vazia. Delete não permitido!\n");
  return '\0';
 }

 if (posicao > l->tamanho || posicao <= 0) { /*II*/
  printf("Posição inválida!\n");
  return '\0';
 }

 if (posicao == 1) { /*III*/

  Elemento *e = l->inicio;
  char caracter = e->valor;
  l->inicio = l->inicio->proximo;

  free(e);

  l->tamanho = l->tamanho - 1;
  return caracter;
 } else {

  Elemento *e_atual = l->inicio; /*IV*/
  Elemento *e_anterior;

  int i; /*V*/
  for(i = 1; i < posicao ; i++){
    e_anterior = e_atual;
    e_atual = e_atual->proximo;
  }
```

```
  e_anterior->proximo = e_atual->proximo; /*VI*/
  char caracter = e_atual->valor;

  free(e_atual);

  l->tamanho = l->tamanho - 1;
  return caracter;
 }
}
```

O código anterior é responsável por:

I. Verificar se a lista é vazia. Se for, não temos como excluir elementos de nenhuma posição;

II. Se não for vazia, precisamos saber se é uma posição válida;

III. Se a posição for `1`, então estamos excluindo o primeiro elemento da lista. Assim, criamos um elemento para armazenar o início a ser excluído (`Elemento *e = l->inicio;`), obtemos o seu valor (`char caracter = e->valor;`) e, por fim, passamos o início da lista para o `proximo` do início excluído (`l->elemento = l->inicio->proximo`). Antes de retornar o valor excluído (`return caracter;`), eliminamos o elemento da memória (`free(e);`) e decrementamos a quantidade de elementos da lista (`l->tamanho = l->tamanho - 1;`);

IV. Entretanto, se não for a posição `1`, temos que encontrar a posição a ser excluída. Para isso, inicialmente criamos um ponteiro auxiliar para começar a navegar na lista a partir de seu início (`Elemento *e_atual = l->elemento;`) e também criamos um ponteiro que auxiliará no momento da exclusão (`Elemento *e_anterior;`), para manter a lista íntegra;

V. Neste ponto, é feita a procura pela posição de exclusão. Para isso, navegamos até acharmos a posição desejada (`for (i = 1; i < posicao ; i++)`). Enquanto essa navegação é realizada, ficamos movimentando os ponteiros auxiliares `e_anterior` e `e_atual` para que

eles se posicionem antes (`e_anterior`) e depois (`e_atual`) da posição a ser excluída;

VI. Quando a posição é atingida, os auxiliares `e_anterior` e `e_atual` ficaram configurados para que o `próximo` do anterior receba o `proximo` do atual (`e_anterior->proximo = e_atual->proximo;`). Isso é necessário para a lista permanecer íntegra. Após isso, o valor excluído deve ser guardado para ser retornado (`char caracter = e_atual->valor;`). O elemento a ser excluído é liberado da memória (`free(e_atual);`) e, por fim, a quantidade de elementos é decrementada em `1` (`l->tamanho = l->tamanho - 1;`) e o valor excluído é retornado (`return caracter;`).

SIZE

```
int size(Lista *l) {
   return l->tamanho; /*I*/
}
```

O código anterior é simples e responsável por realizar a operação `size` , que executa o seguinte passo:

I. Retorna o tamanho da lista `l` , que é armazenado no campo `tamanho` do *struct* **Lista**.

Levando em consideração os três elementos adicionados e o `delete` realizado, a execução desse código exibirá o valor `2` .

Para obter o exemplo completo de nossa lista simplesmente encadeada em C, onde se pode executar (interagindo via prompt) e ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-ListaSimples.git. Lá você conseguirá fazer o download do código e executá-lo.

Duplamente encadeada

Devido a algumas operações permanecerem integralmente iguais ao que já vimos, focaremos em estudar apenas os códigos que possuem mudanças para esse tipo de lista. Assim, a quantidade de operações implementadas nesta seção será menor. Além disso, as codificações da lista dupla são muito similares às da lista simples, precisando apenas de alguns novos ponteiros serem conectados; portanto, apenas essas novas conexões serão explicadas.

Para iniciar nossas implementações, é necessário modificar nossos *structs* e a função de inicialização. Os códigos para isso são, respectivamente:

```
typedef struct elemento {
 char valor;
 struct elemento *proximo;
 struct elemento *anterior;
} Elemento;

typedef struct lista {
 Elemento *inicio;
 Elemento *elemento_corrente;
 int posicao_corrente;
 int tamanho;
} Lista;

Lista* iniciar() {

 Lista *l = malloc(sizeof(Lista));
 l->inicio = NULL;
 l->elemento_corrente = NULL;
 l->tamanho = 0;
 l->posicao_corrente = 0;

 return l;
}
```

Nos códigos anteriores, temos o `struct` chamado **Elemento**, que tem como finalidade aglutinar o dado que a lista armazenará e os ponteiros que serão usados para conectar os elementos da lista nos

dois sentidos (`proximo` e `anterior`). Já o `struct` chamado **Lista** é responsável por definir nossa lista de fato, que é um encadeamento de vários *structs* do tipo **Elemento**. Além disso, esse *struct* também armazena o tamanho atual da lista, sua posição corrente e seu início.

No último código, a função `iniciar()` é responsável por alocar o espaço de memória para o primeiro elemento da lista e iniciá-lo com `NULL` , pois nenhum elemento ainda foi adicionado. Adicionalmente, esse código também configura o tamanho atual da lista com `0` e sua posição corrente também para `0` , pois ela acabou de ser criada. Ao final da configuração, a lista é retornada e está pronta para começar a ser utilizada.

Com a lista definida e iniciada, vamos explorar as operações relevantes.

ADD

```
void add(Lista *l, char caractere) {

 if (empty(l)) {

  Elemento *e = malloc(sizeof(Elemento));
  e->valor = caractere;
  e->proximo = NULL;
  e->anterior = NULL; /*I*/

  l->inicio = e;
 } else {

  Elemento *e = malloc(sizeof(Elemento));
  Elemento *antigo_inicio = l->inicio;

  e->valor = caractere;
  e->proximo = antigo_inicio;

  e->anterior = NULL; /*II*/
  antigo_inicio->anterior = e;
```

```
  l->inicio = e;
 }

 l->tamanho = l->tamanho + 1;
}
```

Assim como na *lista simples*, a *lista dupla* tem sua adição codificada no início, com algumas diferenças.

I. Se a lista está vazia, então estamos inserindo o primeiro elemento. A diferença é que temos que apontar o `anterior` do início para `NULL` ;

II. Caso a lista não esteja vazia, então temos que inserir o novo elemento e "empurrar" o antigo para mais adiante. Além do que já é feito para a *lista simples*, as duas linhas que `II` identifica fazem o apontamento do duplo encadeamento: o `anterior` do novo elemento (que é o novo início) para `NULL` (`e->anterior = NULL;`) e o `anterior` do antigo início para o novo início (`antigo_inicio->anterior = e;`).

ADD com posição

```
void add_pos(Lista *l, char caractere, int posicao) {

 if (empty(l)) {
  printf("Lista vazia. Add posicional não permitido!\n");
  return;
 }

 if (posicao > l->tamanho || posicao <= 0) {
  printf("Posição inválida!\n");
  return;
 }

 if (posicao == 1) {
  add(l, caractere);
  return;
 }
```

```
 Elemento *e_atual = l->inicio;
 Elemento *e_anterior;

 Elemento *e = malloc(sizeof(Elemento));
 e->valor = caractere;

 int i;
 for (i = 1; i < posicao ; i++) {
  e_anterior = e_atual;
  e_atual = e_atual->proximo;
 }

 e_anterior->proximo = e;
 e->proximo = e_atual;

 e->anterior = e_anterior; /*I*/
 e_atual->anterior = e;

 l->tamanho = l->tamanho + 1;
}
```

A diferença do `add_pos` da *lista dupla* para o da *lista simples* é que, aqui, em `I`, temos que conectar o `anterior` do elemento a ser inserido ao elemento anterior à posição de inserção (`e->anterior = e_anterior;`) e conectar o `anterior` do elemento à frente da posição de inserção ao novo elemento inserido (`e_atual->anterior = e;`).

GET

```
char get(Lista *l, int posicao) {

 if (empty(l)) {
  printf("Lista vazia.\n");
  return '\0';
 }

 if (posicao > l->tamanho || posicao <= 0) {
  printf("Posição inválida!\n");
  return '\0';
```

```
}

if (l->elemento_corrente == NULL) { /*I*/
 l->elemento_corrente = l->inicio;
 l->posicao_corrente = 1;
}

if (posicao == l->posicao_corrente) { /*II*/
 return l->elemento_corrente->valor;
} else {

 if (posicao > l->posicao_corrente) { /*III*/

  int i = l->posicao_corrente;
  Elemento *e = l->elemento_corrente;
  while (e != NULL) {

   if (i == posicao) {

    l->elemento_corrente = e; /*IV*/
    l->posicao_corrente = i;
    return e->valor;
   } else {

    e = e->proximo; /*V*/
    i++;
   }
  }
 }

 if (posicao < l->posicao_corrente) { /*VI*/

  int i = l->posicao_corrente;
  Elemento *e = l->elemento_corrente;
  while (e != NULL) {

   if (i == posicao) {

    l->elemento_corrente = e; /*VII*/
```

```
     l->posicao_corrente = i;
     return e->valor;
    } else {

     e = e->anterior; /*VIII*/
     i--;
    }
   }
  }
 }

 return `\0`; /*IX*/
}
```

Em comparação ao `add` e `add_pos` da *lista simples*, que são mais parecidos com o `add` e `add_pos` da *lista dupla*, o `get` na *lista dupla* possui muito mais linhas de código, comparado com o da *lista simples*. Isso acontece porque devemos ficar atualizando e utilizando o `elemento_corrente` e a `posicao_corrente` da lista. Assim, temos:

I. Se estivermos executando o primeiro `get` , começamos sempre do início da lista;

II. Se o `get` atual for igual à execução do `get` anterior, nada há nada a fazer, basta retornar o valor do `elemento_corrente` , no caso, `return l->elemento_corrente->valor;` ;

III. Entretanto, pode não ser igual, então primeiro perguntamos se a posição desejada é maior que a posição atual na lista (`if (posicao > l->posicao_corrente)`). Se for, temos que navegar a partir do elemento corrente (`Elemento *e = l->elemento_corrente;`) no sentido início → fim (esquerda/direita);

IV e V. Para realizar esta navegação, temos que utilizar sempre o `proximo` de cada elemento (`e = e->proximo;`) e incrementar as posições (`i++`). Quando a posição desejada é encontrada, atualizamos o `elemento_corrente` da lista (`l->elemento_corrente = e;`),

sua `posicao_corrente` (`l->posicao_corrente = i;`) e retornamos o valor desejado (`return e->valor;`);

VI, VII e VIII. Toda a lógica explicada para quando a posição desejada for maior que a posição corrente se aplica para quando a posição desejada é menor. Neste caso, o sentido da navegação muda para fim → início (direita/esquerda). Devido a isso, usamos os ponteiros `anterior` e decrementamos as posições (`i--`). As atualizações do `elemento_corrente` e da `posicao_corrente` são iguais;

IX. Se este ponto do código for atingido, é sinal que algo "deu errado" e a posição não foi encontrada. Assim, vazio (`\0`) é retornado.

É válido ressaltar que, como validações de vazio e posição são feitas de forma prévia, é de se esperar que os pontos III ou IV sejam atingidos e um elemento seja retornado. Entretanto, quando se trata de C e ponteiros, uma retorno vazio (em IX) é fornecido para evitar comportamentos adversos, caso a execução atinja esse ponto de forma inesperada.

SET

```c
void set(Lista *l, char caractere, int posicao) {

 if (empty(l)) {
  printf("Lista vazia.\n");
  return;
 }

 if (posicao > l->tamanho || posicao <= 0) {
  printf("Posição inválida!\n");
  return;
 }

 if (l->elemento_corrente == NULL) { /*I*/
  l->elemento_corrente = l->inicio;
  l->posicao_corrente = 1;
 }
```

```
if (posicao == l->posicao_corrente) { /*II*/
 l->elemento_corrente->valor = caractere;
} else {

 if (posicao > l->posicao_corrente) { /*III*/

  int i = l->posicao_corrente;
  Elemento *e = l->elemento_corrente;
  while (e != NULL) {

   if (i == posicao) {

    l->elemento_corrente = e; /*IV*/
    l->posicao_corrente = i;

    e->valor = caractere;
    return;
   } else {

    e = e->proximo; /*V*/
    i++;
   }
  }
 }

 if (posicao < l->posicao_corrente) { /*VI*/

  int i = l->posicao_corrente;
  Elemento *e = l->elemento_corrente;
  while (e != NULL) {

   if (i == posicao) {

    l->elemento_corrente = e; /*VII*/
    l->posicao_corrente = i;

    e->valor = caractere;
    return;
```

```
    } else {

     e = e->anterior; /*VIII*/
     i--;
    }
   }
  }
 }
}
```

O `set` tem um comportamento muito similar ao `get`. Temos apenas uma sutil diferença: em vez de retornarmos o valor do elemento desejado, alteramos o valor do elemento desejado. Assim, temos:

I. Se estivermos executando o primeiro `set`, começamos sempre do início da lista;

II. Se o `set` atual for igual à execução do `set` anterior, não há nada a fazer, basta alterar o valor do `elemento_corrente`, no caso, `l->elemento_corrente->valor = caractere;`;

III. Entretanto, pode não ser igual, então primeiro perguntamos se a posição desejada é maior que a posição atual na lista (`if (posicao > l->posicao_corrente)`). Se for, temos que navegar a partir do elemento corrente (`Elemento *e = l->elemento_corrente;`) no sentido início → fim (esquerda/direita);

IV e V. Para realizar esta navegação, temos que utilizar sempre o `proximo` de cada elemento (`e = e->proximo;`) e incrementar as posições (`i++`). Quando a posição desejada é encontrada, atualizamos o `elemento_corrente` da lista (`l->elemento_corrente = e;`), sua `posicao_corrente` (`l->posicao_corrente = i;`) e modificamos o valor desejado (`e->valor = caractere;`);

VI, VII e VIII. Toda a lógica explicada para quando a posição desejada for maior que a posição corrente se aplica para quando a posição desejada é menor. Neste caso, o sentido da navegação muda para fim → início (direita/esquerda). Devido a isso, usamos os

ponteiros `anterior` e decrementamos as posições (`i--`). As atualizações do `elemento_corrente` e da `posicao_corrente` são iguais.

DELETE

```
char delete(Lista *l, int posicao) {

 if (empty(l)) {
  printf("Lista vazia. Delete não permitido!\n");
  return '\0';
 }

 if (posicao > l->tamanho || posicao <= 0) {
  printf("Posição inválida!\n");
  return '\0';
 }

 if (posicao == 1) { /*I*/

  Elemento *e = l->inicio;
  char caracter = e->valor;
  l->inicio = l->inicio->proximo;
  l->inicio->anterior = NULL;

  free(e);

  l->tamanho = l->tamanho - 1;
  return caracter;
 } else {

  Elemento *e_atual = l->inicio; /*II*/
  Elemento *e_anterior;

  int i;
  for(i = 1; i < posicao ; i++){
    e_anterior = e_atual;
    e_atual = e_atual->proximo;
  }
```

```
  char caracter = e_atual->valor; /*III*/
  e_anterior->proximo = e_atual->proximo;
  e_atual->proximo->anterior = e_anterior;

  free(e_atual);

  l->tamanho = l->tamanho - 1;
  return caracter;
 }
}
```

Ao contrário do `get` e `set`, que têm a capacidade de guardar o elemento corrente, o `delete` não faz isso. Isso porque, após a exclusão, não temos como definir com exatidão qual dos elementos deve se tornar corrente, se o `anterior` ou o `proximo`. Então, sempre iniciamos esse processo do início da lista. Assim, temos:

I. Se a exclusão do elemento for na posição `1` da lista, armazenamos o `inicio` atual da lista e obtemos o valor a ser excluído e posteriormente retornado. Tais operações são, respectivamente, `Elemento *e = l->inicio;` e `char caracter = e->valor;`. Após isso, precisamos fazer o `inicio` da lista passar para o próximo elemento, que será o novo `inicio` da lista. Para isso, fazemos `l->inicio = l->inicio->proximo;`. Nenhum `inicio` de lista tem um elemento anterior, então temos que fazer o `anterior` do novo `inicio` apontar para `NULL` (`l->inicio->anterior = NULL;`). Tendo feito esse processo de movimentação do `inicio` da lista, liberamos da memória o elemento (`free(e);`), decrementamos a quantidade de elementos (`l->tamanho = l->tamanho - 1;`) e, finalmente, retornamos o elemento excluído (`return caracter;`);

II. Entretanto, a posição pode não ser a primeira, então temos que, a partir do início da lista (`Elemento *e_atual = l->inicio;`), procurar a posição desejada. Para isso, fazemos o que já fizemos em códigos anteriores: usamos um contador para procurar a posição desejada e usamos um ponteiro auxiliar (`e_anterior`) para navegar na lista e conseguir mantê-la conectada;

III. No momento em que a posição é encontrada, realizamos os seguintes passos: armazenamos o valor do elemento a ser excluído (`char caracter = e_atual->valor;`), fazemos o `proximo` do elemento anterior apontar para o `proximo` do atual (`e_anterior->proximo = e_atual->proximo;`) e apontamos o `anterior` do `proximo` do elemento atual para o elemento anterior (`e_atual->proximo->anterior = e_anterior;`). Esses três passos são necessários para manter a lista duplamente conectada. Ao fim desse processo, liberamos da memória o elemento (`free(e_atual);`), decrementamos a quantidade de elementos (`l->tamanho = l->tamanho - 1;`) e, finalmente, retornamos o elemento excluído (`return caracter;`).

> Para obter o exemplo completo de nossa lista duplamente encadeada em C, onde se pode executar (interagindo via prompt) e ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-ListaDupla.git. Lá você conseguirá fazer o download do código e executá-lo.

6.5 Exemplos de uso

Da mesma maneira que a pilha e a fila, a lista possui uma grande gama de aplicabilidades dentro da computação e no nosso dia a dia. Vejamos um exemplo a seguir.

Pool de conexões

Várias são as aplicações que usam banco de dados para armazenar suas informações, e elas precisam se conectar a ele. Existem

aplicações que realizam milhares de interações com um banco de dados em questão de poucas horas.

Entretanto, o processo de criar uma conexão com o banco é custoso para uma aplicação. Imagine milhares de conexões: isso poderia se tornar um gargalo para a aplicação e penalizar seu tempo de resposta. É para diminuir esse tempo de criação de conexões que existe o *Pool de conexão*.

Mas o que é um *Pool de conexão*? É isso mesmo, é uma *lista*. Tal lista pode ser gerenciada pela aplicação ou pelo banco de dados, e ela contém várias conexões, que estão prontas para uso. Dessa forma, a aplicação, de forma simultânea, fica obtendo (`get`) as conexões dessa lista de acordo com as requisições que são feitas ao banco de dados. Como não existe uma regra de acesso, cada uma das conexões do *pool* pode ser usada de forma livre por requisições afins. Ao final de seu uso, ela é devolvida (`set`) ao *pool* para que outras requisições possam se utilizar dela. Podemos também configurar para que as conexões sejam eliminadas (`delete`) ou criadas (`add`) de acordo com a demanda por conexões.

6.6 Exercícios

> Todos os exercícios devem usar listas com ponteiros. Use também vetores para armazenar os valores de entrada e a estrutura de repetição `for` .

1) Levando em consideração uma lista inicialmente vazia e que as seguintes operações foram realizadas, responda:

1. `add(l,'%')`
2. `add(l,'!')`
3. `size()`

4. `add_pos(l,'#',2)`
5. `delete(l,1)`
6. `set(l,'@',2)`
7. `get(l, 2)`
8. `add_pos(l,'&',2)`
9. `add_pos(l,'+',1)`
10. `get(3)`
11. `delete(2)`
12. `set(l,'$',3)`
13. `size()`

a) Qual o estado da lista após o passo 2?

b) Qual o estado da lista após o passo 5?

c) Qual o estado da lista após o passo 8?

d) Qual o retorno do passo 10?

e) Qual o estado da lista após o passo 12?

f) Qual o retorno do passo 13?

2) Implemente um programa em C que, a partir da lista simplesmente encadeada de números inteiros `l = {2,6,48,26,1,7,9,0,57,10}`, calcule a soma de seus elementos e informe tal resultado.

3) Implemente um programa em C que utilize uma lista duplamente encadeada de caracteres para armazenar números e verificar se são palíndromos.

Dicas: Os números podem ser fixos no código. Armazene-os separadamente na lista. Use um vetor auxiliar.

4) Implemente um programa em C que utilize uma lista simplesmente encadeada de números inteiros e que não permita que números repetidos sejam adicionados.

Dica: Solicite os valores a serem adicionados.

CAPÍTULO 7
Vetor/Matriz vs. Pilha vs. Fila vs. Lista

Para encerrar os estudos dos capítulos 3, 4, 5 e 6, nos quais vimos as quatro primeiras estruturas de dados (vetor/matriz, pilha, fila e lista), podemos fazer um resumo final. Vimos que essas estruturas tinham algumas pequenas diferenças:

- *Vetores* e *matrizes* são estáticos, ou seja, têm seu tamanho definido em sua criação, e ele não muda. Possuem acesso posicional e são alocados contiguamente na memória;
- *Pilha*, *Fila* e *Lista* são dinâmicas, ou seja, têm seu tamanho variável. São alocadas espaçadamente em endereços de memória livres, já que podem usar ponteiros. Elas também podem ter implementações com vetores, mas há uma perda com essa abordagem, devido à falta de dinamismo de que geralmente essas EDs necessitam.

Embora possuam essas diferenças, uma coisa é comum a todas essas estruturas: a **linearidade**. Os elementos sempre estão ligados de forma sequencial e, no caso de matrizes, podem ter mais de uma dimensão. A única relação que existe entre os elementos dessas estruturas é que eles são conectados em um sentido: início → fim (esquerda/direita), fim → início (direita/esquerda), cima → baixo ou baixo → cima.

Com isso, pode-se pensar: se todas têm essa característica em comum, então qual é a melhor de se usar? Aviso que essa é a pergunta errada! A pergunta certa é: Qual problema precisamos resolver? A partir disso é que podemos escolher a estrutura de dados linear que se adeque ao problema que se deseja solucionar. Não existe uma ED linear melhor que outra. Cada uma tem um porquê de sua existência e um nicho de problemas no qual se encaixa melhor para ajudar a resolvê-los.

Entender de forma clara o comportamento de cada uma ajuda a escolher a que mais se adequa ao problema a ser resolvido. Algumas dicas ajudam nessa escolha:

- Se tem a certeza de que a quantidade de elementos é fixa, use um **vetor**. Assim, o acesso será mais rápido e se consumirá menos memória, pois não precisará de ponteiros. Se precisar de mais de uma dimensão, use uma **matriz**;
- Se é necessário manter uma sequência dos elementos em que os últimos elementos devem ser acessados primeiro, use uma **pilha**;
- Precisa criar prioridades entre os elementos? Use uma **fila**;
- Apenas quer ter uma disposição linear dos elementos e precisa de acesso aleatório a eles? Use a **lista**.

Podemos ver que cada ED tem uma regra para seu uso muito bem definida. Lembre-se disso! Antes de encerrar o estudo das estruturas lineares, um fato pode ser destacado. Se o notou, parabéns! Se não, tudo bem. Este livro foi feito para lhe ajudar a aprender mais e a notar o que será dito a seguir:

> A verdade é que **PILHAS** e **filas** são **listas**!

Calma, não se desespere. Mas isso é verdade mesmo. Se notarmos, principalmente quando implementamos uma pilha ou uma fila com ponteiros, elas ficam **muito** similares a uma lista. O que de fato torna uma pilha e uma fila diferente de uma lista é que pilhas e filas possuem uma política bem definida em relação à sua manipulação. Isso mesmo que você deve estar pensando: LIFO e FIFO! Pilha e fila devem seguir obrigatoriamente essas políticas para poderem ser manipuladas da forma adequada. Já uma lista é livre em sua manipulação.

CAPÍTULO 8
Árvore

Nos capítulos de 3 a 6, vimos as estruturas de dados lineares e como elas são muito úteis e amplamente utilizadas na computação. Entretanto, existem situações em que não é suficiente os dados estarem dispostos linearmente.

É necessário que exista uma relação mais forte entre eles, que os elementos estejam subordinados ou relacionados entre si. Ou seja, é necessária uma relação *hierárquica* ou *relacional*.

É para suprir essas novas necessidades que existem duas novas estruturas de dados: **árvore** e **grafo**. Nestes dois próximos capítulos, estudaremos essas estruturas, a começar pelas *árvores*.

É válido ressaltar que, para manusear tais estruturas, precisaremos usar *recursividade*. Caso necessite reforçar ou conhecer esse conceito, aconselho a leitura do ***Apêndice IV: Recursividade***.

8.1 Fundamentos

Podemos definir uma **árvore** como:

> Uma estrutura de dados onde os elementos possuem um vínculo do tipo *pai* e *filho*, que cria uma relação hierárquica de subordinação entres eles.

A definição anterior reforça o que foi dito no início deste capítulo: que as árvores são estruturas hierárquicas, pois têm um relacionamento de subordinação entre os elementos no qual *pais*

podem possuir *filhos* e tais *filhos* deve pertencer a **somente um *pai***.

Geralmente, os *filhos* podem ser categorizados em "filhos mais à esquerda" ou "filhos mais à direita". A depender do tipo de árvore, essa categorização se torna mais evidente. Existe ainda uma pequena restrição relativa a essa relação, que veremos na seção *Terminologias*, onde veremos vários termos pertinentes a essa ED.

Um exemplo clássico desse tipo de estrutura em nosso dia a dia é a *árvore genealógica* de uma família. Essa árvore representa todos os membros pertencentes a uma determinada família, levando em consideração um membro inicial. A imagem a seguir ilustra a árvore genealógica de Dom Pedro I.

Dom Pedro I

Maria Amélia

Maria II

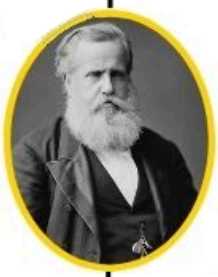
Miguel

João Carlos

Januária

Paula de Bragança

Francisca



Dom Pedro II

Dom Afonso

Princesa Isabel

Leopoldina

Dom Pedro Afonso

Luisa Vitória

Pedro de Alcântara

Luis Maria

Antônio

Figura 8.1: Árvore genealógica de Dom Pedro I.

Através da árvore anterior, podemos detectar que Dom Pedro II é um *filho* de Dom Pedro I, pois o primeiro está subordinado ao segundo. Nesse caso, Dom Pedro I é o elemento *pai* do elemento Dom Pedro II. A mesma ideia se aplica à Princesa Isabel, que é um elemento *filho* de Dom Pedro II, seu elemento pai. Isso demonstra que podemos ter "árvores dentro de árvores", basta mudarmos o ponto de referência.

Terminologias

Por ser uma ED um pouco mais complexa, a árvore possui um conjunto de termos que auxiliam seu entendimento e uso. Levemos em consideração a imagem a seguir, que é a forma mais comum de se representar uma árvore. Esta é uma árvore simples, na qual apenas a regra essencial dessa ED deve ser considerada: *pais podem possuir filhos ou não*. A seguir, veremos algumas das terminologias referentes a essa ED.

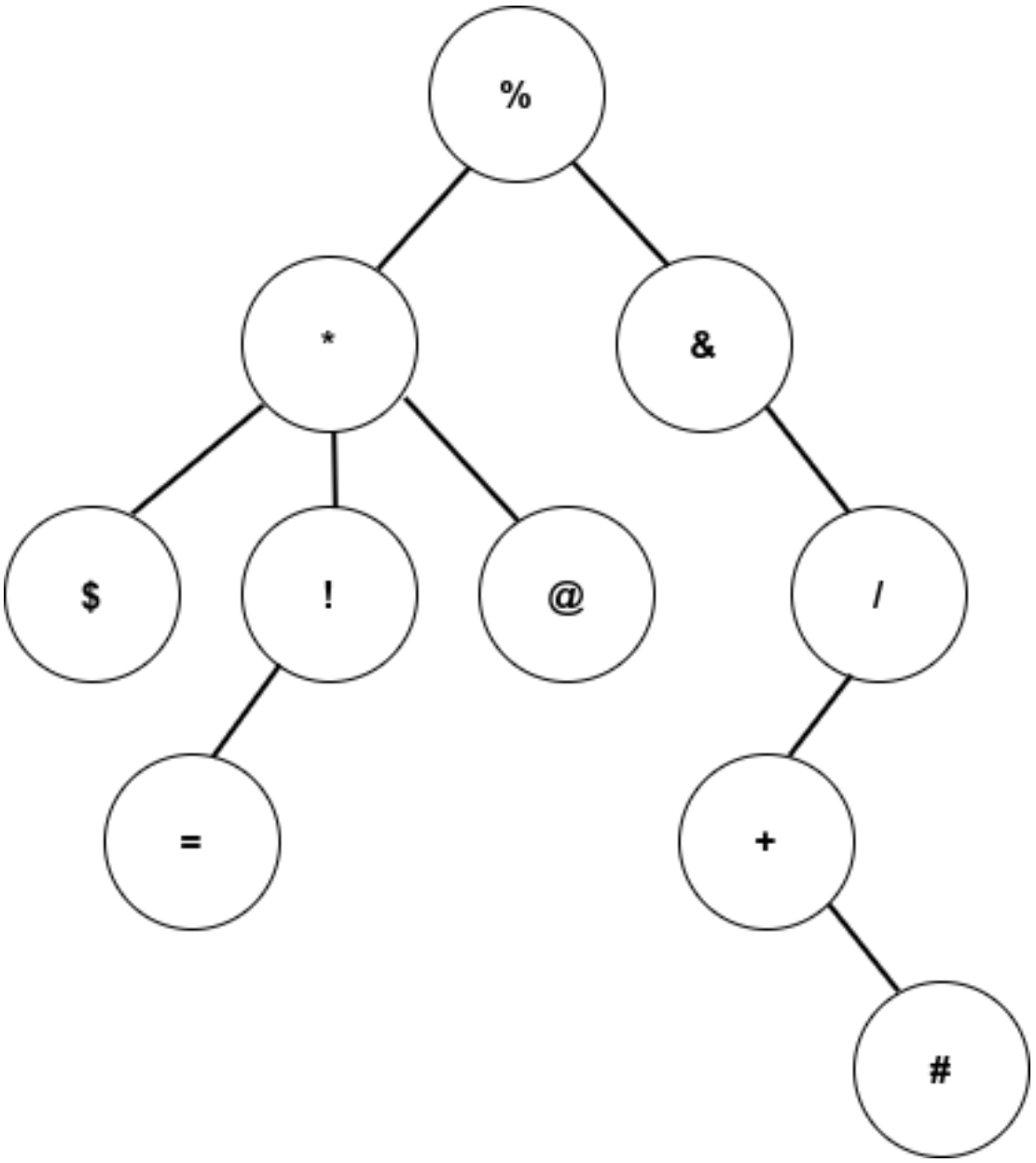


Figura 8.2: Árvore.

- **Raiz (*Root*)**: é o elemento (*nó*) inicial de uma árvore. É a partir dele que ela se inicia e por ele que se inicia o processo de percorrê-la. Observação importante: embora também seja um nó, este é o único que não possui um *nó pai*. A raiz da árvore de exemplo é `%`;
- **Nó (*Node*)**: é o nome dado ao elemento que constitui uma árvore. Ele é constituído por um valor e conectores (geralmente

ponteiros) para outros nós. Todos os círculos em nossa árvore de exemplos são nós;

- **Nó interno**: é um nó que *possui pelo menos um "filho"*. Ou seja, são **"nós pais"**. `*`, `&`, `!`, `/` e `+` são nós internos. Embora `%` seja um "nó pai", ele não se encaixa nessa definição por ser a *raiz*;
- **Nó externo (*Leaf*)**: é um nó que *não possui "filhos"*. Nesse caso, são "nós filhos". O termo mais usado para nomear esse tipo de nó é **nó folha**. Nós folhas sempre se encontram ao final de uma árvore. `$`, `=`, `@` e `#` são nós folhas;
- **Ancestral**: são os nós que *antecedem* um determinado nó. Isso muda de acordo com o nó em questão. Levando em consideração o nó `=`, seus ancestrais são `!`, `*`, `%`. Já levando em consideração `*`, temos somente `%` como ancestral — no caso, a *raiz*;
- **Descendente**: são os nós que *procedem* de um determinado nó. Isso muda de acordo com o nó em questão. Levando em consideração o nó `*`, seus descendentes são `$`, `!`, `@`, `=`. Se levarmos em consideração `%`, teremos todos os demais nós da árvore;
- **Grau do nó (*Degree*)**: é a quantidade de "filhos" que um determinado nó possui. Levando em consideração o nó `*`, temos um grau `3`, que são `$`, `!` e `@`. Levando em consideração o nó `/`, temos um grau `1`, que é `+`;
- **Grau da árvore (*Degree of tree*)**: é a maior quantidade de "filhos" de um determinado nó. Em nossa árvore, o grau é `3`, pois esse é maior grau de um dos seus nós — no caso, `*`;
- **Profundidade (*Depth*)**: é a quantidade de nós a partir da *raiz* até a *folha* mais distante. Em nossa árvore, temos uma profundidade de `5`, pois vamos de `%` até `#`, passando por `&`, `/` e `+`. Cada nó visitado para se obter a profundidade é chamado de **nível** (*Level*). A depender da necessidade, podemos também usar o termo **profundidade** para *nós internos*;
- **Largura (*Width*)**: é a quantidade de nós de um determinado nível. No nível 3 de nossa árvore, temos a largura `4` — no caso,

os nós `$`, `!`, `@` e `/`. Essa largura pode variar de acordo com o *nível*, e nossa árvore demonstra isso;
- **Largura da árvore (*Width of tree*)**: é a maior largura de nós de um determinado nível da árvore. Em nossa árvore, a largura é `4`, embora tenhamos larguras `1` e `2`;
- **Floresta (*Forest*)**: são subárvores dentro de uma árvore principal. Em nossa árvore, temos *florestas* começando em `*`, `&`, `!`, `/` e `+`;
- **Tamanho da árvore (*Size of tree*)**: é a quantidade total de nós que a árvore possui. Nossa árvore tem um tamanho de `10` nós.

Outras representações

Embora a representação de uma árvore como a da *Figura 8.2* seja a mais difundida, ainda existem outras formas de a representar. A representação gráfica com círculos e linhas é de fato visualmente muito mais intuitiva e amigável, entretanto, programaticamente, criar estas estruturas gráficas é bem mais complexo. Devido a isso, outras duas opções de representação mais fáceis de codificar podem ser utilizadas: parentização e indentação.

Na representação com parênteses, cada nó pai pode possuir um par de parênteses, abrindo e fechando, que englobam seus nós filhos, caso existam. Levando em consideração a árvore da *Figura 8.2*, sua representação com parênteses seria: `%(*($!(=)@)&(/(+(#))))`. Aí vemos que `$`, `=`, `@` e `#` são nós folhas, pois não possuem o par de parênteses. Já `%`, `!`, `&`, `/` e `+` são nós pais, pois possuem tais parênteses. Vale ressaltar que o primeiro nó dessa representação é a raiz da árvore, no caso, `%`.

Na representação com indentação, cada nó pai estará um nível acima de seus filhos e mais à esquerda. Consequentemente, os filhos estarão um nível abaixo e mais à direita. Novamente, levando em consideração a árvore da *Figura 8.2*, sua representação com indentação seria:

```
%
.*
..$
..!
...=
..@
.&
../
...+
....#
```

Na representação anterior vemos novamente que `$`, `=`, `@` e `#` são nós folhas, pois não possuem nós subordinados a si. Já `%`, `!`, `&`, `/` e `+` são nós pais, pois cada um possui seus nós subordinados. Vale ressaltar que o nó mais acima nesta representação é a raiz da árvore, no caso, `%`.

8.2 Operações

Assim como as estruturas lineares (pilha, fila e lista), árvores também possuem as operações de **Inserir**, **Pesquisar**, **Atualizar** e **Excluir**. É importante ressaltar que essas operações podem variar seus comportamentos de acordo com o tipo de árvore, mas, em sua essência, se comportam da seguinte forma:

- `INSERT` : adiciona um novo nó filho em um nó pai — `insert(raiz, pai, valor)`;
- `SEARCH` : obtém um nó da árvore — `search(raiz, valor)`;
- `UPDATE` : modifica um nó da árvore — `update(raiz, nó, valor)`;
- `DELETE` : exclui um nó da árvore — `delete(raiz, valor)`.

Para que essas quatro operações possam ser executadas em uma árvore, é preciso *navegar* por ela. Para realizar isso, é necessário passar pelos seus nós a partir de uma das três formas existentes de navegação: **Pré-ordem**, **Pós-ordem** e **Em-ordem**.

Dessa forma, podemos então detectar que, diferentemente das EDs anteriormente estudadas, que eram lineares, as árvores possuem um processo de acesso aos seus elementos (*nós*) mais aprimorado por ser hierárquica, pois não apenas temos a ideia de *cima-baixo* ou *esquerda-direita*, mas, sim, uma ordem de navegação pelos elementos (nós) que, a depender da forma utilizada, levará a um diferente resultado de exibição. É válido informar que essas navegações podem ser *à esquerda* ou *à direta*, e que uma não é melhor do que a outra, sendo apenas uma questão referencial. Todavia, a navegação *à esquerda* é universalmente a mais utilizada e é a adotada neste livro. Essas formas de navegação são apresentadas a seguir.

Pré-ordem (*pre-order*)

Nessa forma de percorrer os nós de uma árvore, inicia-se sua exibição a partir da *raiz* e segue-se exibindo (visitando) primeiramente sempre um *nó interno*. Somente após esses nós internos serem visitados é que se visita um *nó filho* e/ou um *nó folha*.

Levando em consideração nossa árvore de exemplo da Figura 8.2, teríamos o seguinte resultado ao percorrê-la em *pré-ordem* com navegação *à esquerda*: `%`, `*`, `$`, `!`, `=`, `@`, `&`, `/`, `+` e `#`. A imagem a seguir demonstra, conceitualmente, esse percurso.

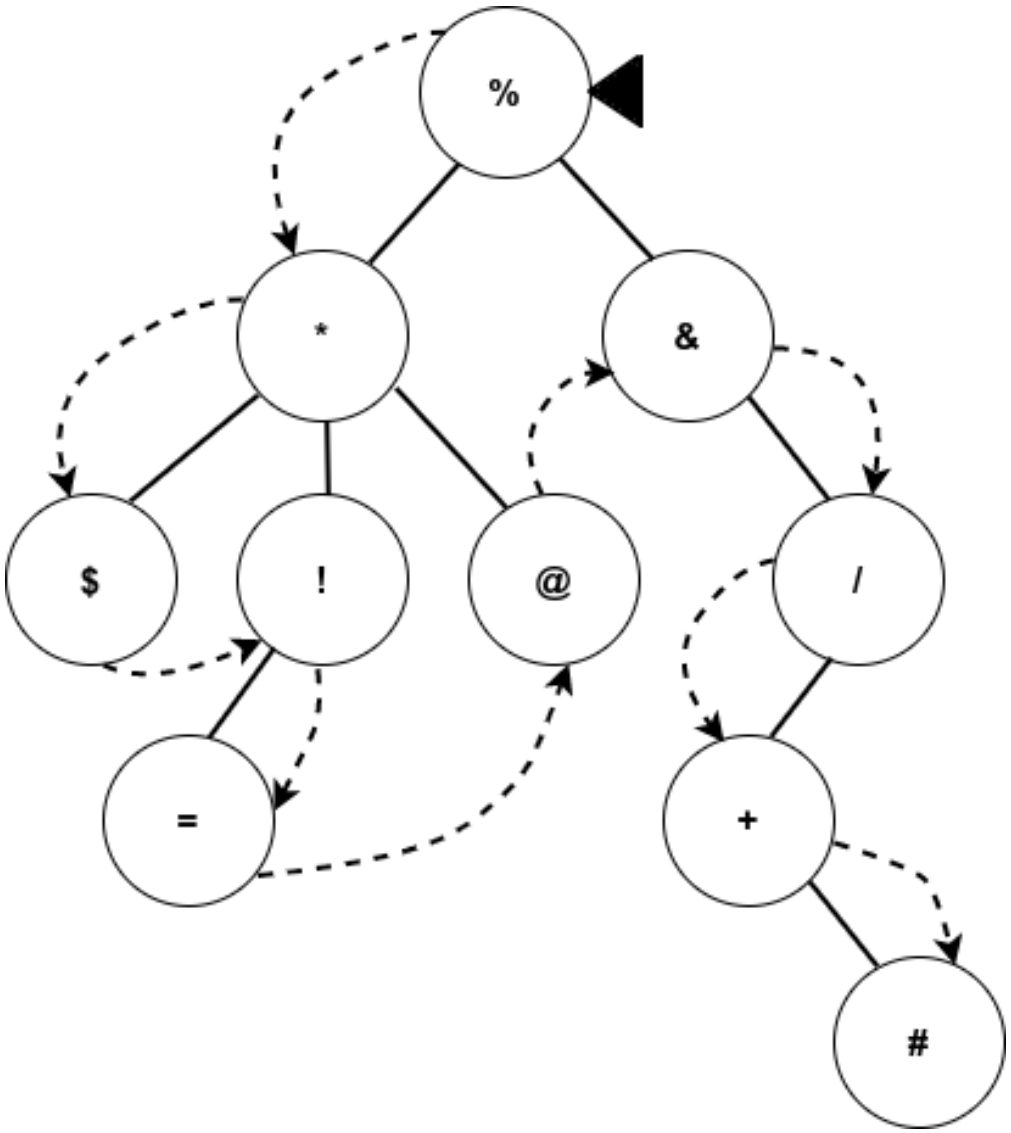


Figura 8.3: Árvore percorrida em pré-ordem (conceitualmente).

Embora a imagem anterior seja o resultado de um percorrimento em *pré-ordem*, sabemos que, para chegar a um nó folha de fato, sempre devemos passar pelo nó interno (*pai*). Assim, o percurso que realmente é executado é o da imagem a seguir. Nele, nota-se que alguns nós são visitados mais de uma vez (o nó `*`, por exemplo, é visitado 4 vezes) para atingir o resultado desejado.

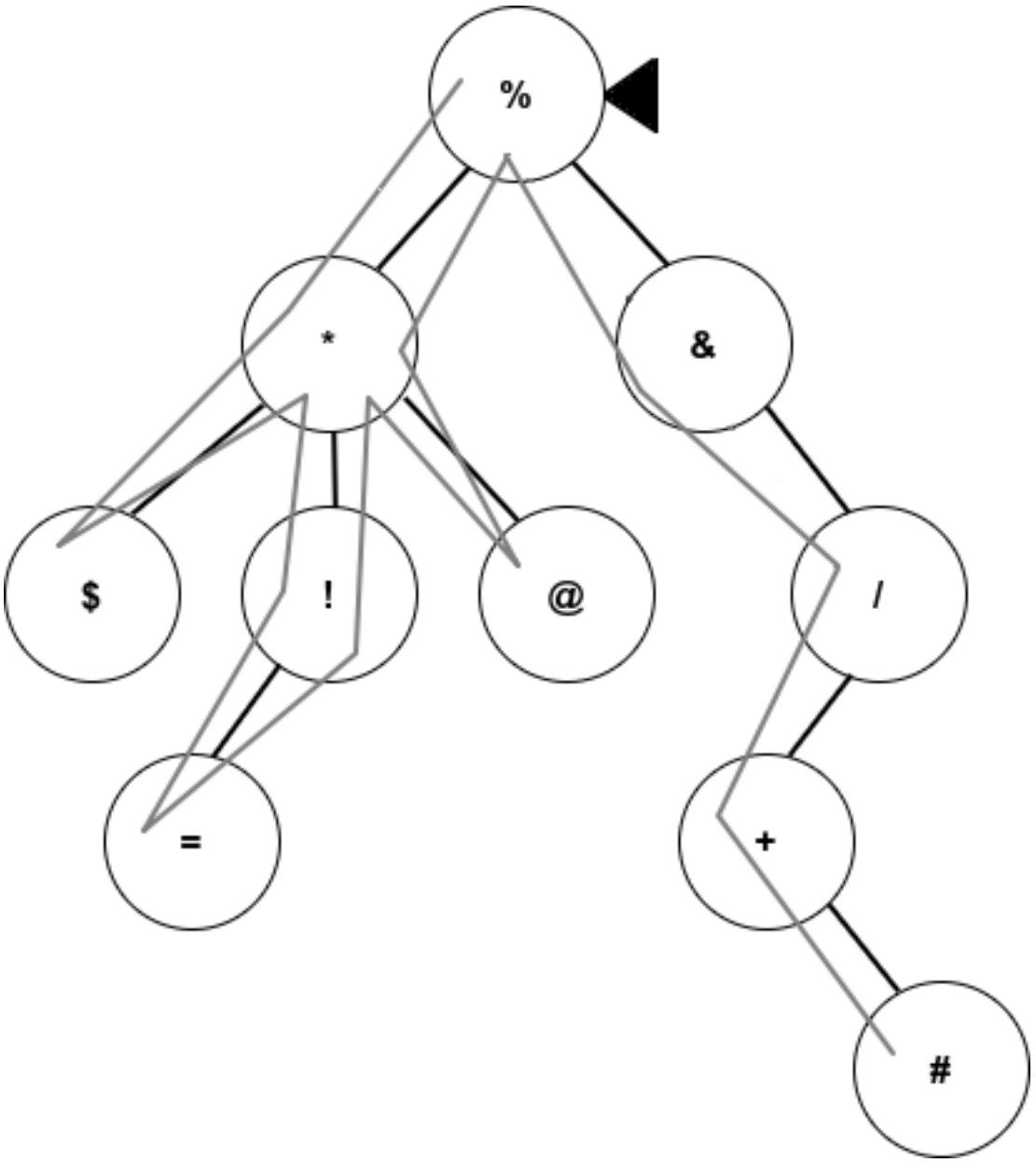


Figura 8.4: Árvore percorrida em pré-ordem (percurso real).

Pós-ordem (*post-order*)

Nessa forma de percorrer os nós de uma árvore, inicia-se sua exibição sempre a partir do *nó folha* mais distante e segue-se exibindo (visitando) primeiramente sempre esses nós folhas e depois os nós internos. Somente após visitar todos eles é que a *raiz* é exibida.

Levando em consideração nossa árvore de exemplo da Figura 8.2, teríamos o seguinte resultado ao percorrê-la em *pós-ordem* com navegação *à esquerda*: `$`, `=`, `!`, `@`, `*`, `#`, `+`, `/`, `&` e `%`. A imagem a seguir demonstra, conceitualmente, esse percurso. Nota-se que, quando não existe um *nó folha* mais *à esquerda*, começa-se pelo primeiro disponível — no caso, o nó `#`.

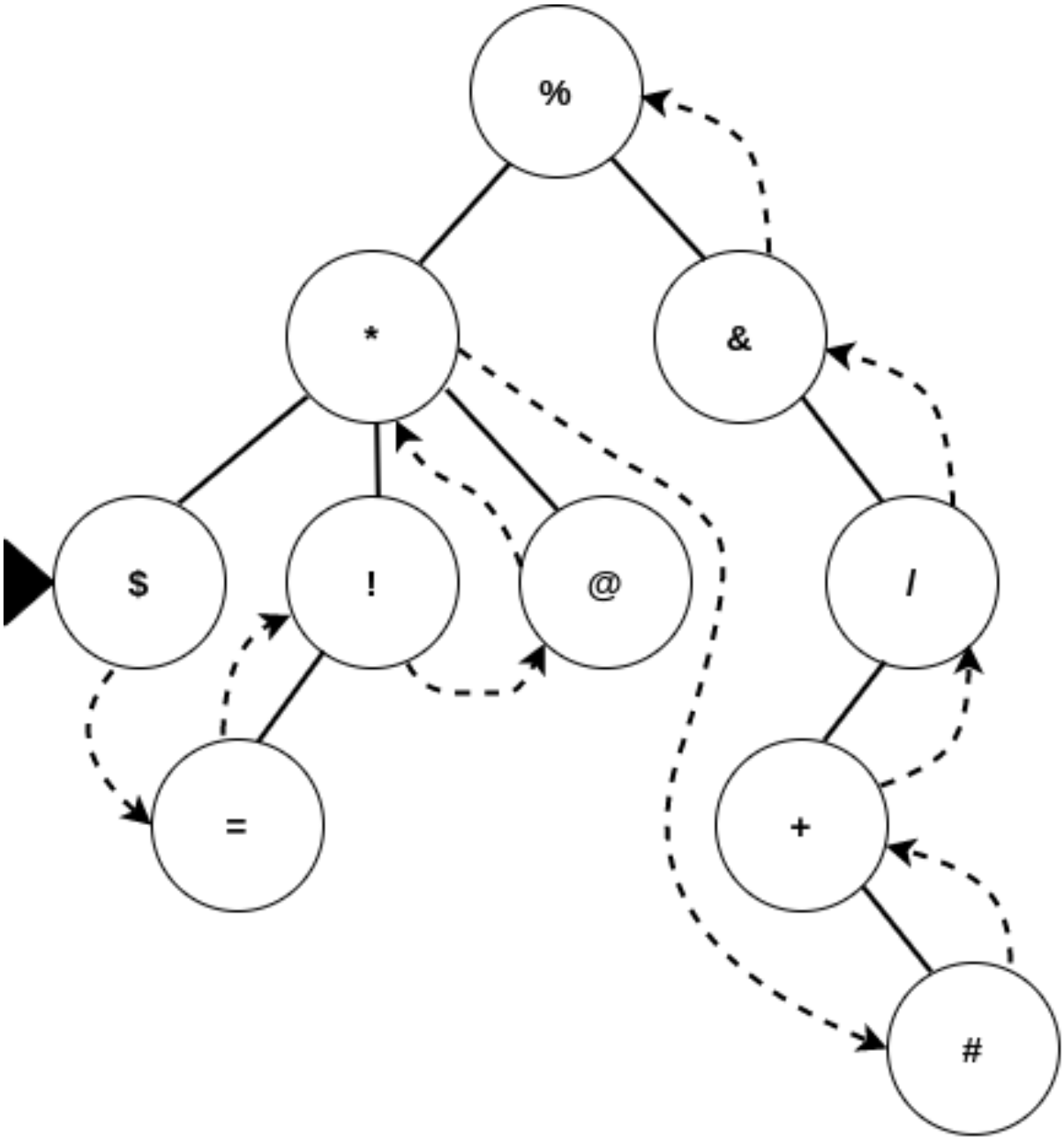


Figura 8.5: Árvore percorrida em pós-ordem (conceitualmente).

Novamente, existe um percurso real executado que é apresentado na imagem a seguir. Mais uma vez, alguns nós são visitados repetidamente (o nó `*`, por exemplo, foi visitado 3 vezes) para atingir o resultado desejado.

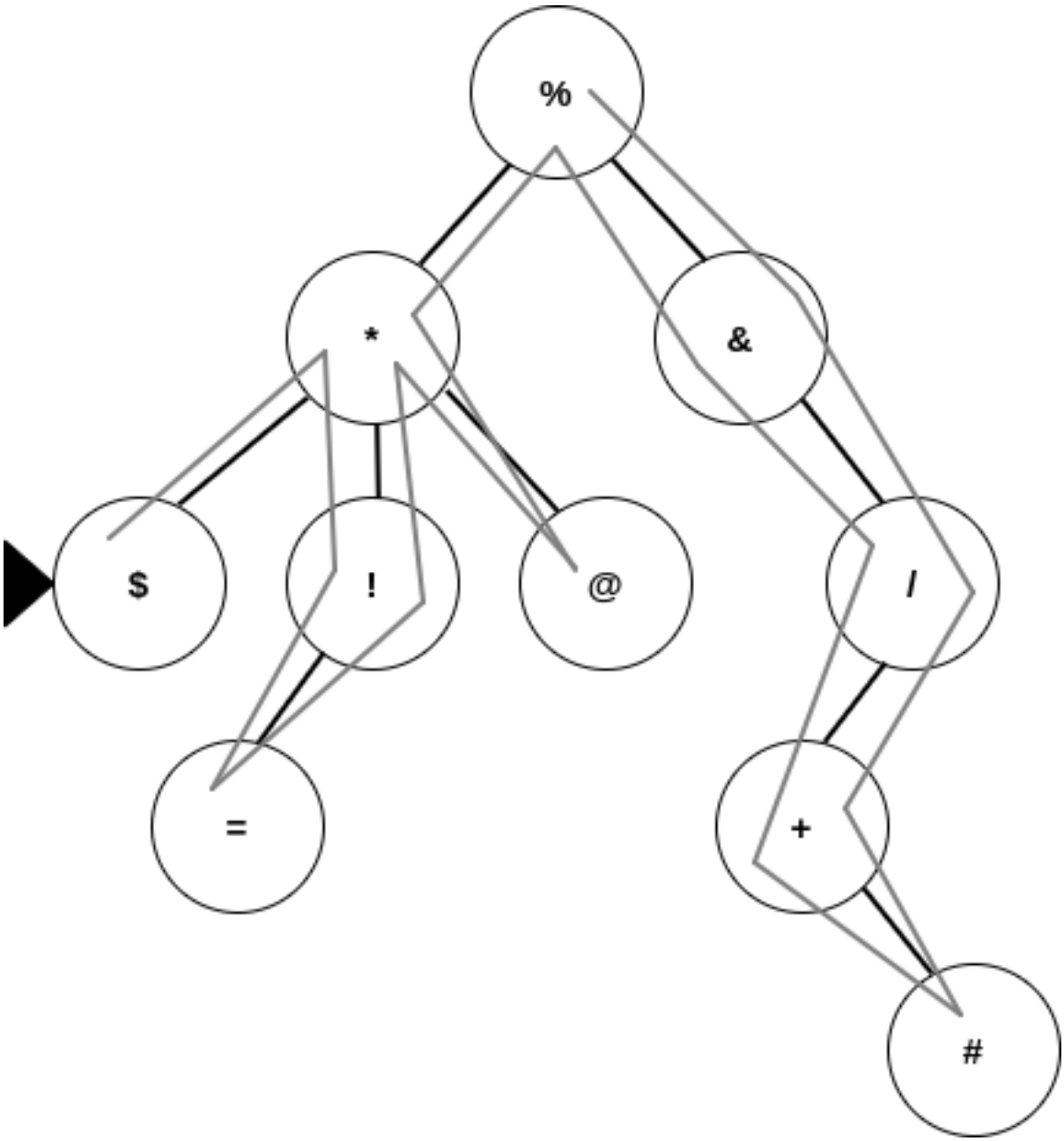


Figura 8.6: Árvore percorrida em pós-ordem (percurso real).

Em-ordem (*in-order*)

Nessa terceira e última forma de percorrer uma árvore, inicia-se sua exibição sempre a partir do *nó folha* mais distante e segue-se exibindo "na ordem" de visitação dos nós, sejam eles *nós folhas* e/ou *nós internos*. Após uma subárvore ser completamente visitada, a *raiz* é exibida e navega-se para a outra subárvore e se repete o processo.

Ainda levando em consideração nossa árvore de exemplo da Figura 8.2, teríamos o seguinte resultado ao percorrê-la *em-ordem* com navegação *à esquerda*: `$`, `*`, `=`, `!`, `@`, `%`, `&`, `+`, `#` e `/`. A imagem a seguir demonstra, conceitualmente, esse percurso. Nota-se que, quando não existe um *nó folha* mais *à esquerda*, começa-se pelo primeiro disponível, no caso o nó `&` e `+`.

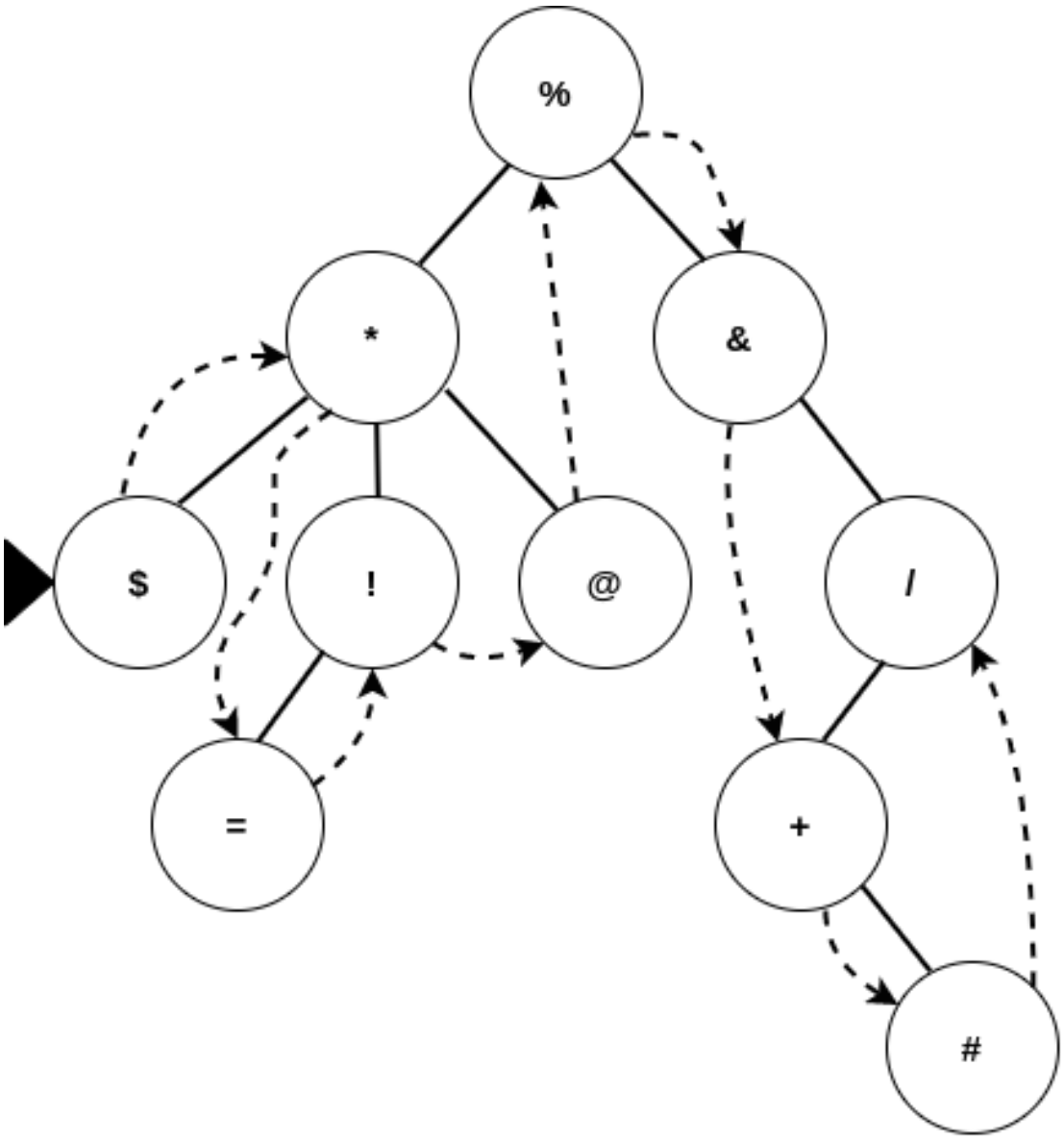


Figura 8.7: Árvore percorrida em-ordem (conceitualmente).

Mais uma vez, existe um percurso real, apresentado na imagem a seguir. Novamente, alguns nós são visitados de forma repetida (como exemplo o nó `*`, que é visitado 2 vezes).

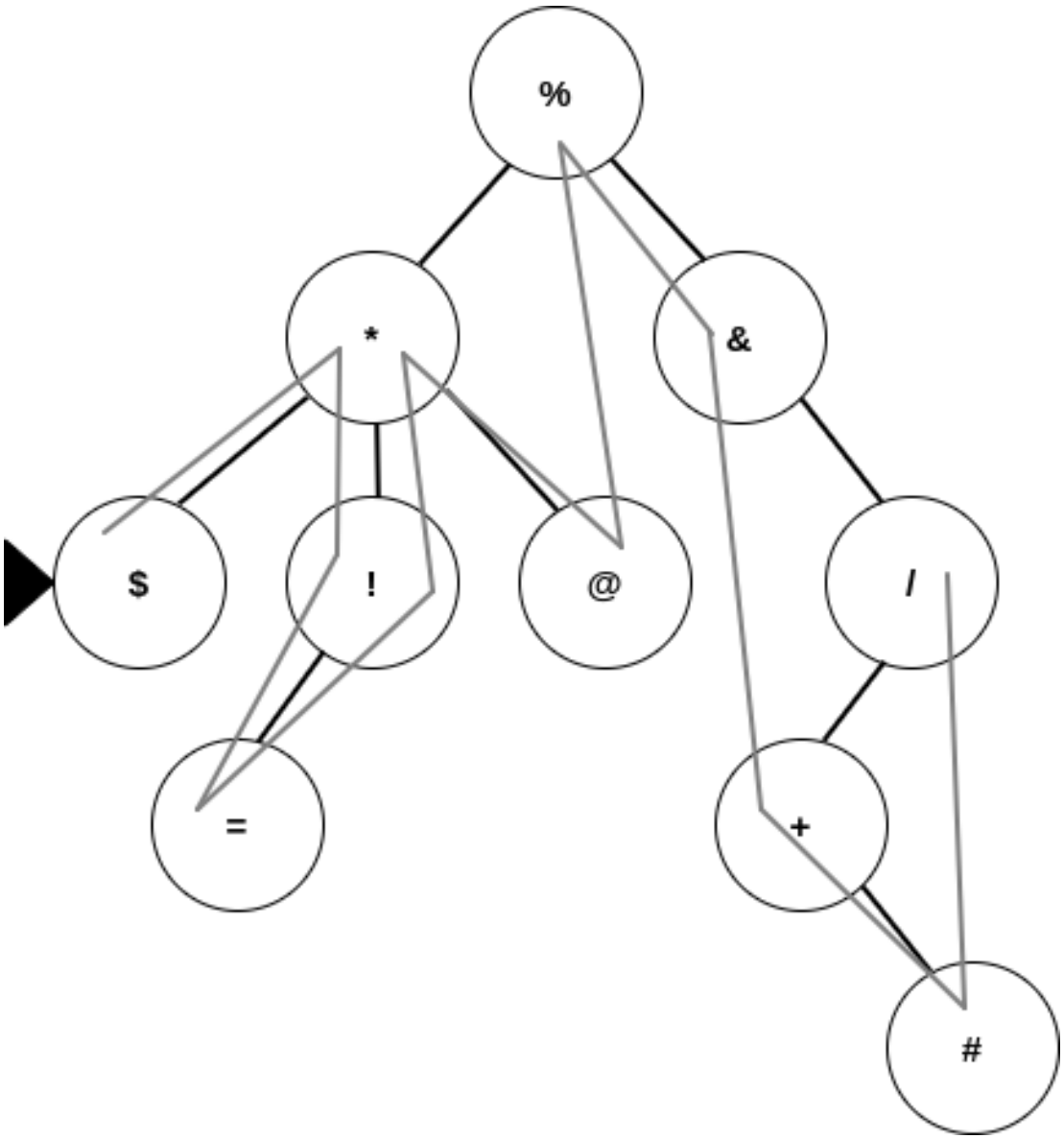


Figura 8.8: Árvore percorrida em-ordem (percurso real).

Apenas para complementar, embora tenhamos também adotado a navegação *à esquerda*, a listagem a seguir apresenta como seria a navegação *à direita* para nossa árvore de exemplo:

- Pré-ordem: `%`, `&`, `/`, `+`, `#`, `*`, `@`, `!`, `=` e `$`;
- Pós-ordem: `#`, `+`, `/`, `&`, `@`, `=`, `!`, `$`, `*` e `%`;
- Em-ordem: `/`, `#`, `+`, `&`, `%`, `@`, `*`, `!`, `=` e `%`.

8.3 Tipos de árvores

Dependendo de como os elementos estão organizados dentro da árvore e da quantidade de nós filhos que os nós pais tenham, podemos identificar vários tipos de árvores. Existem dezenas de tipos de árvores, então aqui exploraremos as mais comumente usadas e que são essenciais para o entendimento dessa ED.

Clássica

A árvore do tipo **clássica**é a mais fácil de trabalhar e entender. Ela é somente a disposição dos nós de forma hierárquica, sem nenhuma restrição de ordenação ou quantidade de nós filhos. A nossa árvore da Figura 8.2 é um exemplo desse tipo.

Binária

A **árvore binária** é um tipo no qual a quantidade de *nós filhos* deve ser exatamente `0`, `1` ou `2`. A figura a seguir ilustra essa característica.

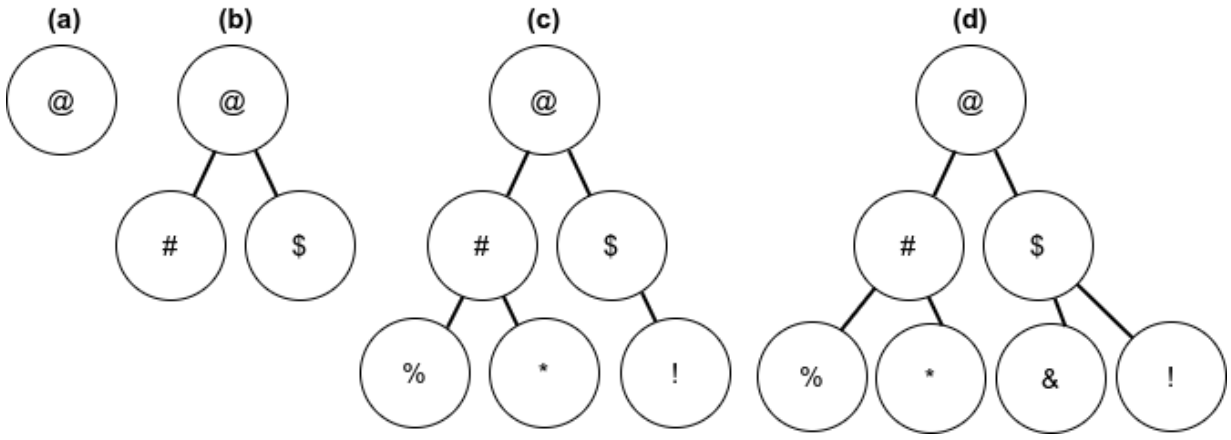


Figura 8.9: Árvore binária.

Em `(a)`, verificamos que o nó raiz possui `0` filhos. É a árvore mais simples que se pode ter. É válido ressaltar que uma árvore que possui somente o *nó raiz* é válida em qualquer tipo de árvore.

Já em `(b)`, temos uma árvore binária em que o nó `@` possui exatamente `2` filhos: `#` e `$`. Em `(c)`, o nó `#` possui `2` filhos (`%` e `*`), mas o nó `$` possui somente `1` filho (`!`). Por fim, na árvore `(d)`, todos os nós, em exceção dos nós folhas, possuem exatamente `2` filhos. Quando a situação ilustrada em `(b)` e `(d)` se configura, temos o que é chamado de **árvore binária completa**.

Binária de busca

A **árvore binária de busca** é uma variação da **árvore binária** que, além de atender à regra de quantidade de *nós filhos* desta, é necessariamente ordenada. Sobre ordenação de árvores, haverá uma seção mais adiante que explicará como atingir esse comportamento. Além disso, toda árvore **AVL** é uma **árvore binária de busca** — e a **AVL** será explicada detalhadamente na seção a seguir. Devido a esses dois fatores, não será necessário explorar a fundo esse tipo de árvore.

AVL

A **árvore AVL** (árvore Adelson-Velskii e Landis) é uma variação da *árvore binária de busca* na qual, tendo como referência qualquer um de seus nós, a diferença entre a **profundidade** de sua subárvore à esquerda e de sua subárvore à direita é de, no máximo, `1` — no caso, `|prof_e(nó) - prof_d(nó) | < 2`. Quando essa situação ocorre, dizemos que a árvore AVL é uma *árvore balanceada*. Para ilustrar essa situação, levemos em consideração a imagem a seguir:

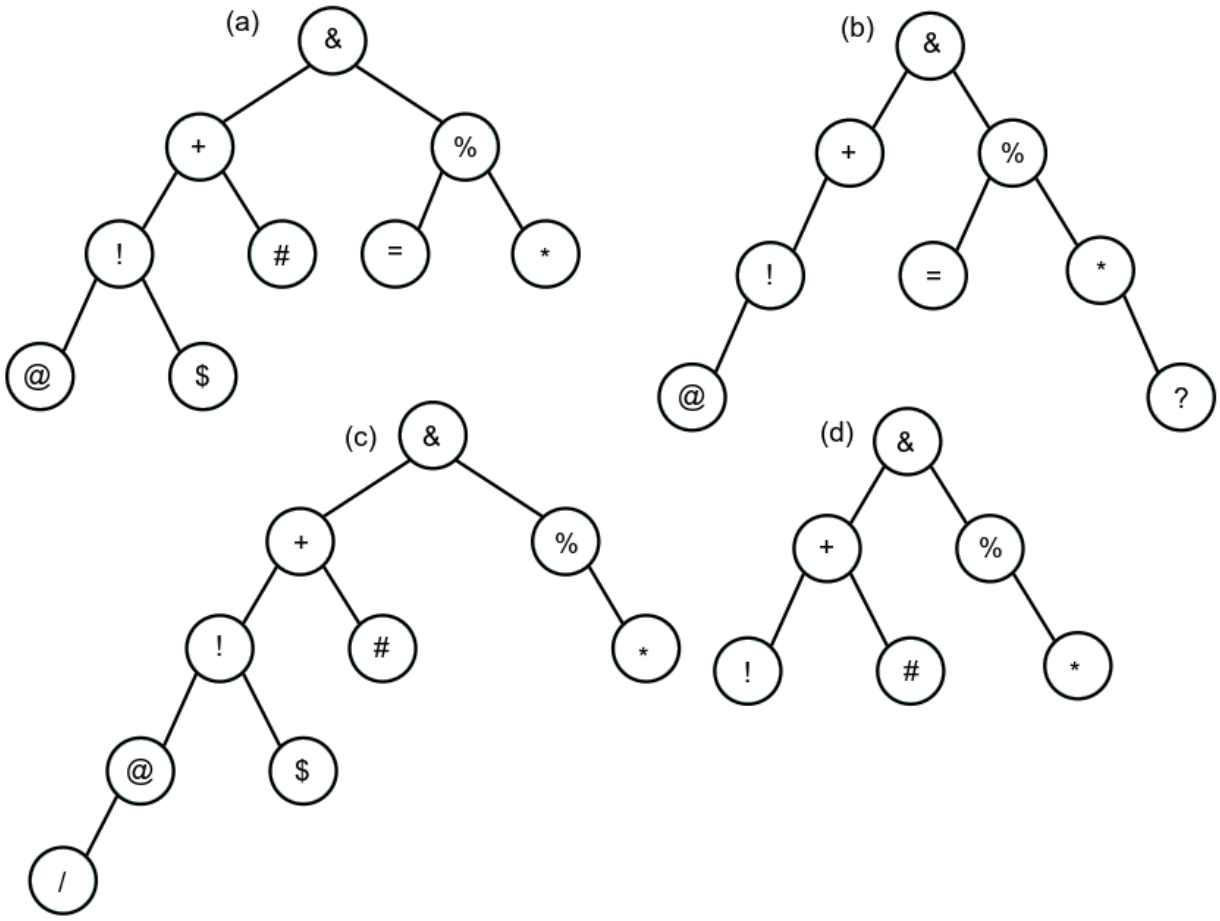


Figura 8.10: Árvore AVL (balanceada).

Na imagem anterior, todas as árvores podem ser classificadas como *árvores binárias*, mas somente `(a)` e `(d)` são AVL, pois em nenhum nó destas duas temos uma diferença de profundidade entre suas subárvores maior que `1`. Entretanto, podemos notar que `(b)` e `(c)` não são AVL, pois existem nós com uma diferença de profundidade maior que `1` — por exemplo, o nó `+`. Como exemplo, podemos usar a árvore `(c)`: `|prof_e(nó) - prof_d(nó) | < 2` ⟶ `|prof_e(+) - prof_d(+) | < 2` ⟶ `|3 - 1| < 2` ⟶ `|2| < 2` ⟶ `2 < 2`, o que é falso.

Levando em consideração a fórmula `|prof_e(nó) - prof_d(nó) | < 2`, algumas informações são relevantes:

- Se a subárvore à esquerda estiver mais profunda que à direita, o resultado é `+1`;

- Se as subárvores estiverem com a mesma profundidade, o resultado é `0`;
- Se a subárvore à direita estiver mais profunda que à esquerda, o resultado é `-1`;
- Se a subárvore à esquerda está desbalanceada para um nó `N`, o resultado é `> 1`;
- Se a subárvore à direita está desbalanceada para um nó `N`, o resultado é `< -1`.

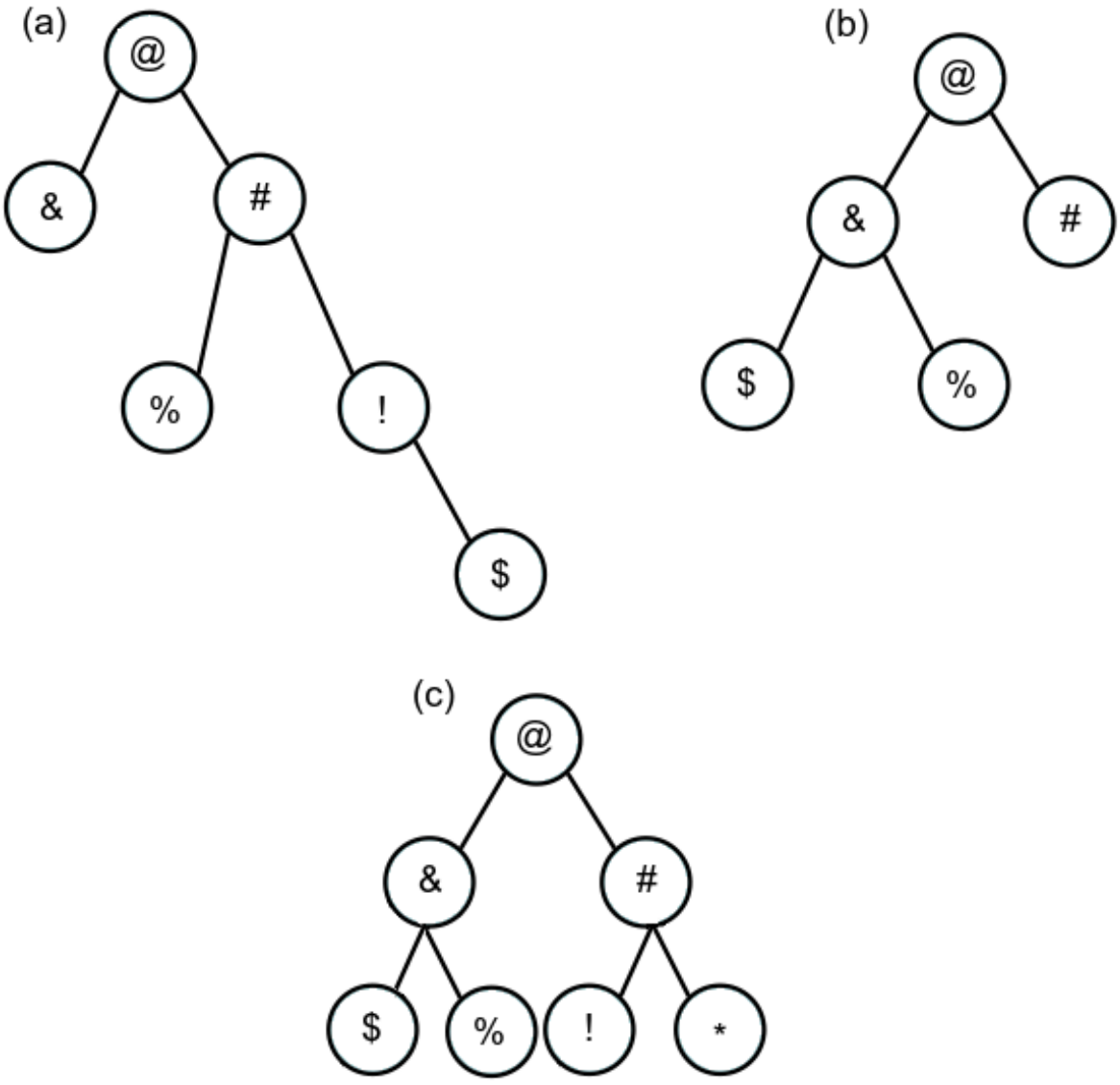


Figura 8.11: Exemplos de desbalanceamentos e balanceamento.

Em `(a)`, levando em consideração o nó `@`, temos `|prof_e(@) - prof_d(@) | < 2 ⟶ |1 - 3| < 2 ⟶ |-2| < 2 ⟶ 2 < 2`. Assim, o resultado foi `-2`, ou seja, `< -1`, o que prova o desbalanceamento para a direita. Se avaliarmos o nó `@` da árvore `(b)`, temos: `|prof_e(@) - prof_d(@) | < 2 ⟶ |2 - 1| < 2 ⟶ |1| < 2 ⟶ 1 < 2`. No caso, o resultado foi `+1`, o que prova que a árvore à esquerda é mais profunda, mas que ainda não existe desbalanceamento. Por fim, em `(c)` temos uma árvore totalmente balanceada, pois se aplicarmos a fórmula em qualquer uma de suas subárvores, seja à esquerda ou à direita, o resultado será `0`.

É válido ressaltar que, quando desbalanceamentos ocorrem, devido a inserções ou exclusões, é necessário reorganizar a árvore para que ela ainda possa ser considerada AVL. Assim, devem ser realizadas *movimentações* nos nós, com certas regras. Estas movimentações são chamadas de **rotações** e têm a finalidade de deixar a árvore em uma organização que gere sempre o mesmo resultado, após inserções ou exclusões, quando for percorrida na navegação *em-ordem*.

Existe um conjunto de rotações que podem ser aplicadas para que a árvore se mantenha balanceada e, consequentemente, considerada *AVL*:

- Rotação simples à esquerda;
- Rotação simples à direita;
- Rotação dupla à esquerda;
- Rotação dupla à direita.

A seguir, vamos explorá-las em detalhes.

Rotação simples à esquerda

Essa rotação deve ser realizada quando a árvore estiver *desbalanceada para a direita*. O seguinte algoritmo deve ser aplicado:

1. Separe toda a subárvore à direita;

2. Retire toda a subárvore à esquerda da árvore separada e conecte-a no local da árvore original, da qual a subárvore à direita foi retirada;
3. Torne toda a árvore original (modificada anteriormente) uma subárvore à esquerda da árvore separada.

Ao aplicarmos a fórmula `|prof_e(nó) - prof_d(nó)|` na árvore a seguir, obtemos o valor `-2` para o nó `@`. Ou seja, ela está desbalanceada para a direita; logo, a rotação deve ser para a esquerda. Os passos acima são representados na imagem a seguir:

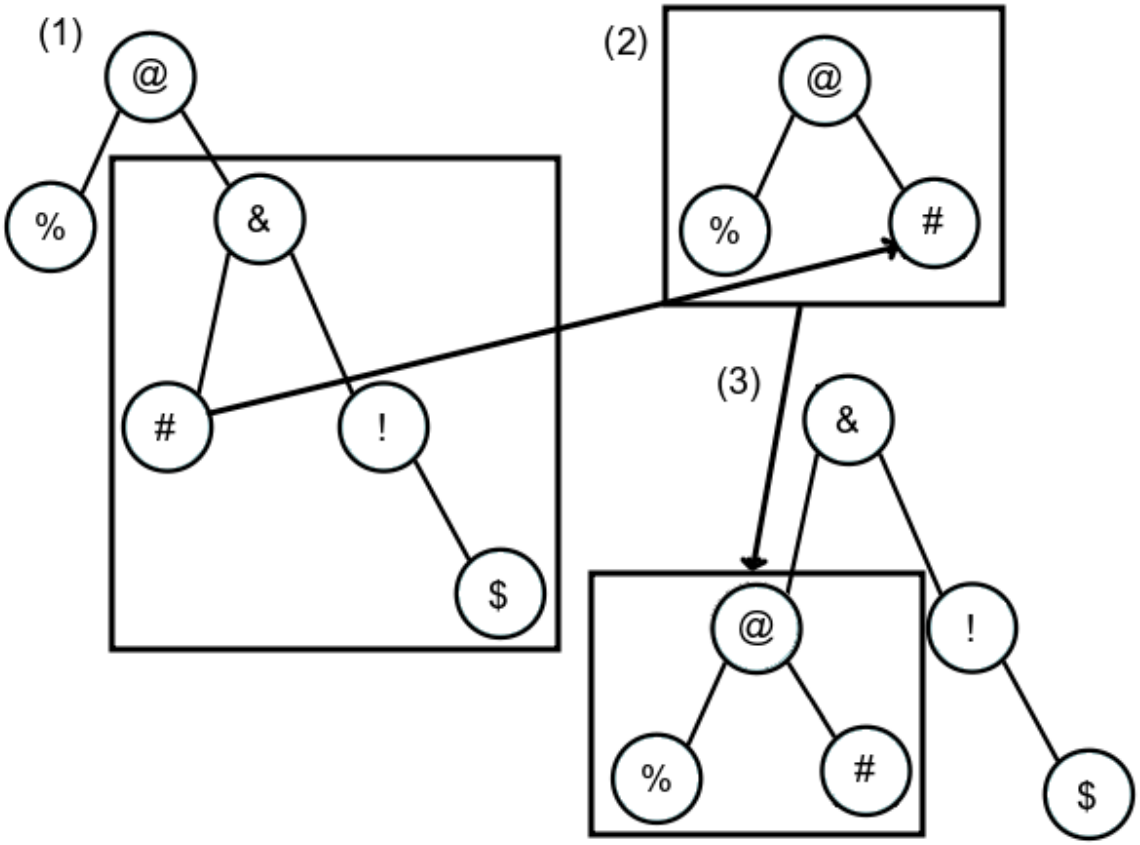


Figura 8.12: Rotação simples à esquerda.

Para finalizar, podemos destacar dois pontos:

- Todos os nós subordinados a `%` e `#` devem ser movimentados na íntegra e sem modificações;
- A navegação *em-ordem* para `(1)` e `(3)` é `%, @, #, &, $` e `!`.

Esses dois pontos destacados podem ser aplicados para as rotações a seguir.

Rotação simples à direita

Essa rotação deve ser realizada quando a árvore estiver *desbalanceada para a esquerda*. O seguinte algoritmo deve ser aplicado:

1. Separe toda a subárvore à esquerda;
2. Retire toda a subárvore à direita da árvore separada e conecte-a no local da árvore original da qual a subárvore à esquerda foi retirada;
3. Torne toda a árvore original (modificada anteriormente) uma subárvore à direita da árvore separada.

Ao aplicarmos a fórmula `|prof_e(nó) - prof_d(nó)|` na árvore a seguir, obtemos o valor `+2` para o nó `@`. Ou seja, ela está desbalanceada para a esquerda, logo, a rotação deve ser para a direita. Os passos descritos anteriormente são representados na imagem a seguir:

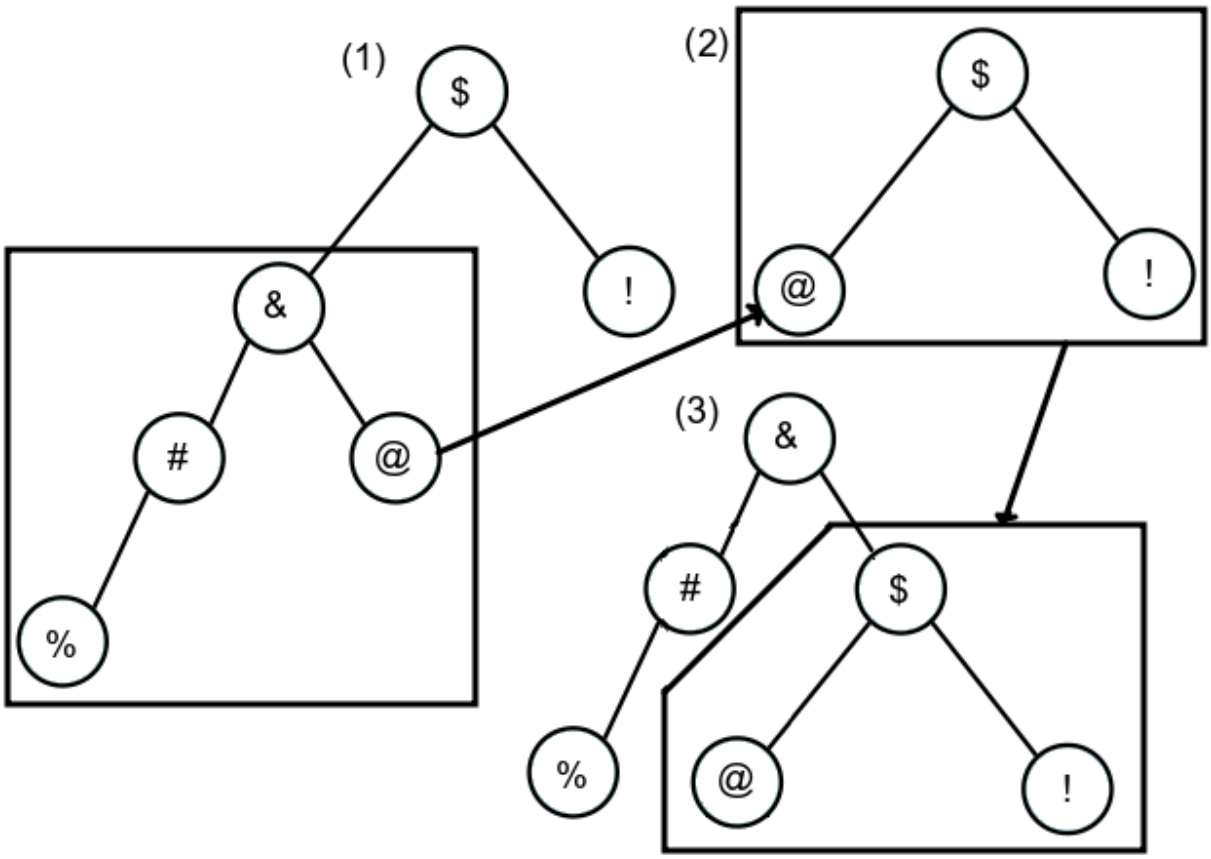


Figura 8.13: Rotação simples à direita.

Rotação dupla à esquerda (Esquerda-Direita)

Essa rotação deve ser realizada quando a árvore estiver *desbalanceada para a direita e com um desvio para a esquerda*, o que forma um espécie de quina. Essa rotação executa os seguintes passos:

1. Aplica-se uma **rotação simples à direita** na subárvore à direita do nó desbalanceado;
2. Aplica-se uma **rotação simples à esquerda** no nó desbalanceado da árvore resultante do passo 1.

Ao aplicarmos a fórmula `|prof_e(nó) - prof_d(nó)|` na árvore a seguir, obtemos o valor `-2` para o nó `#`. Ou seja, ela está desbalanceada para a direita, mas com uma *subárvore* à esquerda mais profunda.

Logo, a rotação deve ser dupla para a esquerda. A aplicação dos passos anteriores é representada na imagem a seguir, com duas figuras para $_1$ e para $_2$.

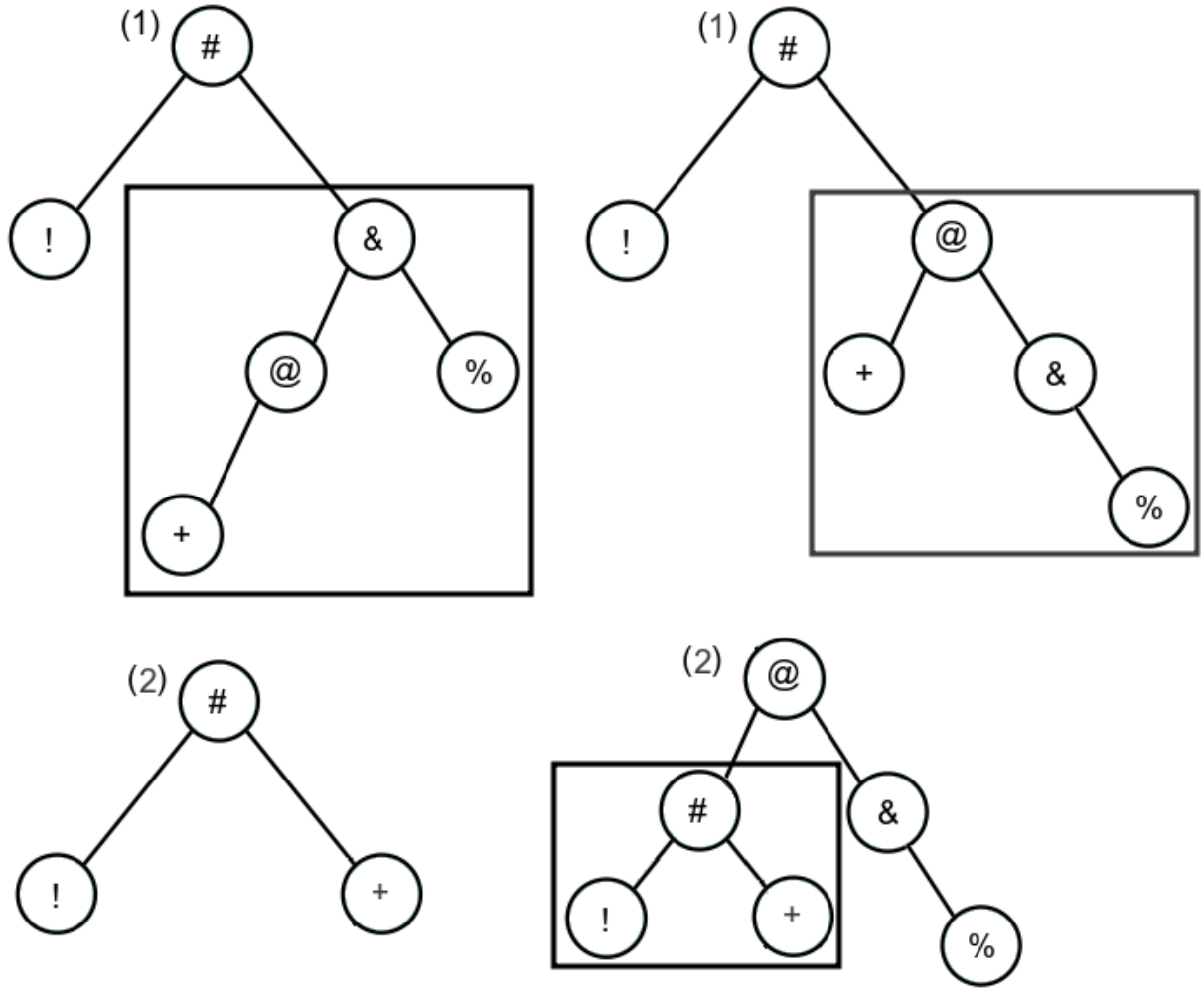


Figura 8.14: Rotação dupla à esquerda.

Rotação dupla à direita (Direita-Esquerda)

Essa rotação deve ser realizada quando a árvore estiver *desbalanceada para a esquerda e com um desvio para a direita*. Essa rotação executa os seguintes passos:

1. Aplica-se uma **rotação simples à esquerda** na subárvore à esquerda do nó desbalanceado;

2. Aplica-se uma **rotação simples à direita** no nó desbalanceado da árvore resultante do passo 1.

Ao aplicarmos a fórmula `|prof_e(nó) - prof_d(nó)|` na árvore a seguir, obtemos o valor `+2` para o nó `&`. Ou seja, ela está desbalanceada para a esquerda, mas com uma *subárvore* à direita mais profunda. Logo, a rotação deve ser dupla para a direita. A aplicação dos passos acima é representada na imagem a seguir, com duas figuras para `1` e para `2`.

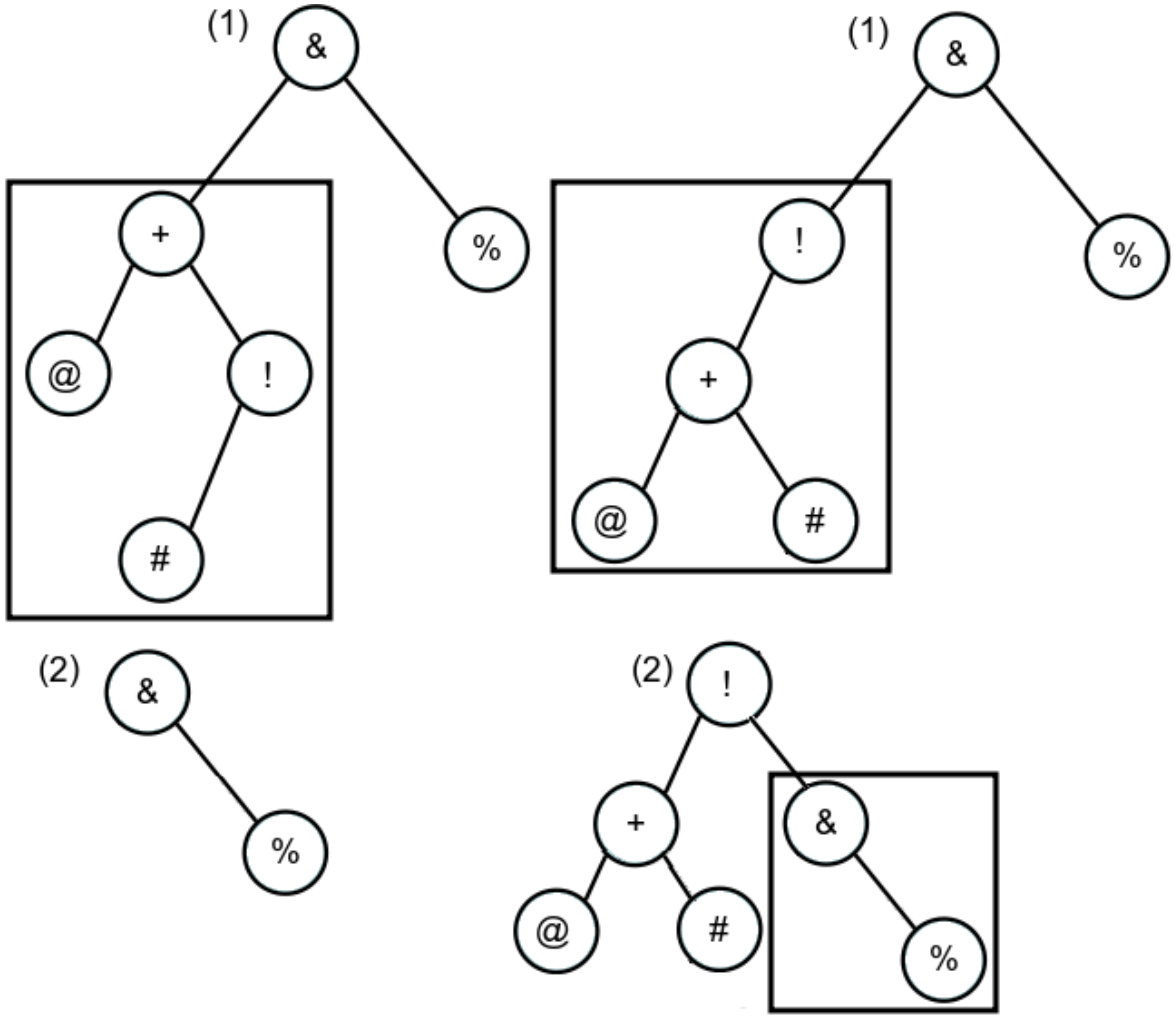


Figura 8.15: Rotação dupla à direita.

N-árias

Nesse tipo de árvore, não existe nenhuma limitação na quantidade de *nós filhos*. Ou seja, podemos ter um nó pai com cinco filhos, outro com um filho, outro com oito. Cada nó tem a quantidade de filhos que precisar. Mas há duas ressalvas:

- Preferencialmente, a árvore deve possuir uma largura **muito** maior que sua profundidade;
- Obrigatoriamente, todo nó sempre terá o primeiro **descendente** à esquerda, e os demais serão irmãos dele.

Em relação à primeira observação, podemos dizer que esse tipo é como um oposto a *árvores binárias/AVL* que têm `0` a, no máximo, `2` nós filhos. Além disso, podemos verificar que essa quantidade maior de filhos deixa a árvore mais rasa e larga, ao contrário das binárias/AVL, que são profundas e estreitas.

Já em relação à segunda observação, vemos que esse tipo de árvore tem uma restrição que as binárias/AVL não têm: o primeiro filho é sempre à esquerda, ao contrário das binárias/AVL, que podem ter um primeiro filho à direita. Isso é devido aos filhos de um nó estarem ligados diretamente aos seus irmãos e não ao seu pai. Essa diferença, embora possa parecer pequena, traz consigo uma mudança significativa na implementação. Para demonstrar melhor essas diferenças, vejamos uma *árvore binária* e uma possível representação dela como *N-ária*.

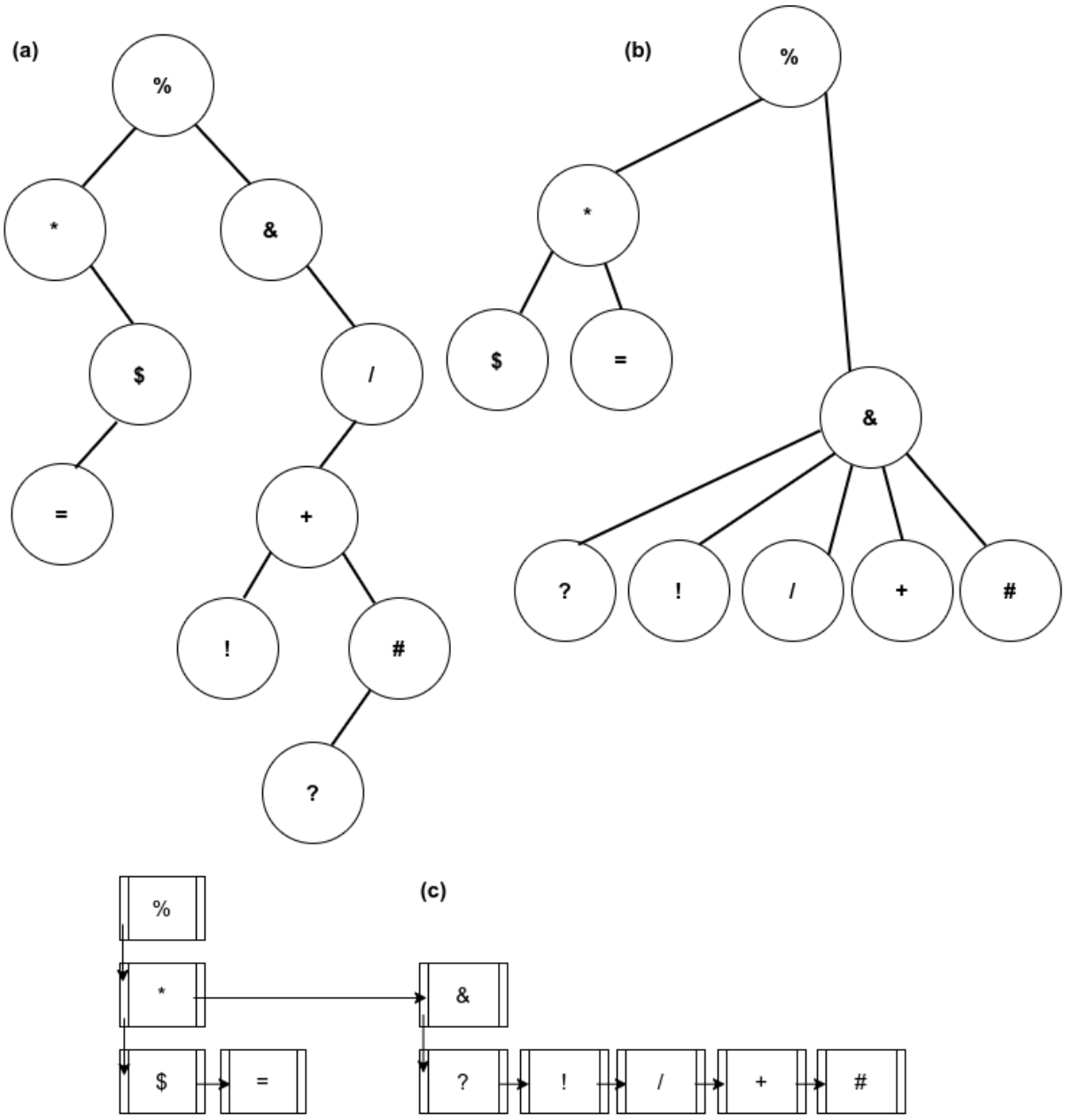


Figura 8.16: Binária -> N-ária.

Em (a), temos uma *árvore binária*, com nós tendo um ou dois filhos. Alguns desses filhos estão unicamente à esquerda e outros à direita. Essa árvore tem profundidade 5 e largura 2. Em (b), temos uma representação em *árvore N-ária* que possuiria uma relação hierárquica similar à da árvore binária. Podemos notar que ela tem profundidade 2 e largura 5.

A figura `(b)` implica na representação de ponteiros apresentada em `(c)`. Nota-se que, dos nós filhos, somente os mais à esquerda se conectam diretamente ao pai. Os demais são irmãos (mais à direita), que se conectam entre si.

Por fim, nesse tipo de árvore, é possível que o nó armazene mais de um valor se for necessário. Também é possível que *irmãos* possuam *subárvores*. Entretanto, neste livro abordaremos a versão mais simples: nó com um valor e irmãos com subárvores.

Rubro-negras (RN)

As **árvores rubro-negras** são uma variação das árvores AVL. Devido a isso, elas possuem todas as mesmas propriedades e operações das AVL, mas com as seguintes propriedades a mais:

- Todo *nó* da árvore deve ser negro (preto) ou rubro (vermelho);
- A *raiz* é sempre negra;
- *Nós folhas* são sempre negros e são "filhos" nulos, mas que fazem parte da árvore;
- Nós *rubros* possuem filhos *negros*;
- A partir de qualquer nó para qualquer um de seus nós folhas descendentes, existe o mesmo número de *nós pretos* para qualquer caminho percorrido.

A imagem a seguir ilustra essas propriedades e demonstra como é uma árvore desse tipo.

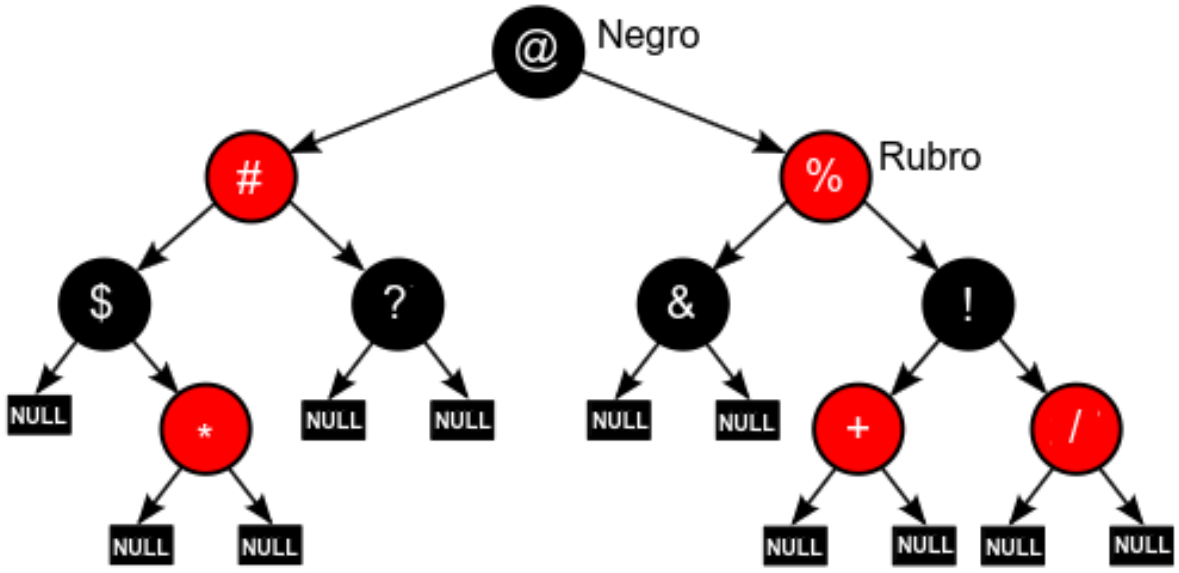


Figura 8.17: Árvore Rubro-Negra.

Na imagem anterior, notamos que a raiz `@` é negra. Os nós folhas são todos representados por `null` e são negros, ou seja, `?` e `&` não são folhas, mas, sim, nós internos. Se filhos forem adicionados a eles, tais filhos se comportarão como `*`, `+` e `/`. Os nós `#` e `%`, que são rubros, possuem filhos negros.

A aplicação dessa propriedade adicional permite que a árvore RN esteja quase sempre balanceada, evitando uma quantidade maior de rotações. O que possibilita isso são os filhos nulos, que fazem parte da árvore. Dessa maneira, mesmo que eles não possuam um dado em seu nó, estão presentes somente para manter a altura da árvore balanceada. Em contrapartida, as AVL são totalmente balanceadas e seus nós sempre devem possuir dados. Isso termina por requerer muito mais rotações para manter a árvore balanceada.

Outros tipos de árvores

Como dito, existem dezenas de tipos de árvores e que, muitas vezes, têm aplicabilidades bem específicas. Todavia, a grande maioria

desses tipos são variações dos tipos aqui apresentados, que sofrem modificações para tratar problemas de forma mais eficaz. Como exemplos, podemos citar:

- Árvores B e B+;
- Árvore Heap;
- Árvore Trie;
- Árvore H.

Embora não seja considerado uma referência oficial, o link a seguir, da Wikipédia, fornece uma vasta lista de tipos de árvores. Caso precise de algo bem específico, a lista auxiliará no pontapé inicial para a resolução de problemas: https://en.wikipedia.org/wiki/Category:Trees_(data_structures)

Por que tantos tipos de árvores?

Essa é uma dúvida comum de se ter e um pouco dela já foi respondido na seção anterior:

> *Alguns tipos de árvores possuem aplicabilidades bem específicas*.

Um bom exemplo destas especificidades são os tipos *AVL* e *rubro-negras*. Estas árvores são muito similares: a RN é uma variação da AVL. Como vimos, as AVL devem sempre estar balanceadas e, para isso, rotações são constantemente realizadas. Ou seja, inserções e remoções são mais custosas nesse tipo de árvore, pois toda vez que uma destas é realizada, rotações talvez precisem ser realizadas para manter o balanceamento. Mas isso traz um benefício: as pesquisas são rápidas. Já a RN, por sempre estar "quase totalmente balanceada", requer menos rotações. No caso, inserções e remoções são menos custosas, mas, em contrapartida, pesquisas são mais lentas do que nas AVL.

Então, qual dessas é a melhor? Na verdade, essa pergunta está errada! A pergunta adequada é: **Qual devo usar para resolver**

meu problema?

Se tivermos uma grande quantidade de dados com inserções e remoções constantes, a AVL é ruim, pois será muito custoso manter a árvore balanceada, e isso pode se transformar em um gargalo. Nesse caso, é melhor usar uma RN, que tem uma performance melhor para inserções e remoções — lembrando que, nesta, a pesquisa é mais lenta. Entretanto, se as inserções e remoções não são constantes e a pesquisa deve ser uma operação rápida, a AVL é e melhor opção.

Outra comparação pode ser feita entre as árvores binárias e árvores N-Árias. Devido às binárias só poderem possuir no máximo dois nós filhos, eventualmente elas podem ficar muito profundas, ou seja, com uma altura muito grande. Isso pode ser um problema a depender da quantidade de nós, pois várias chamadas recursivas serão feitas, e isso pode terminar por comprometer o consumo de memória. Como possível solução, o uso de N-Árias pode auxiliar no tocante ao menor consumo de memória, pois elas não têm limitação na quantidade de nós filhos, o que leva a um menor consumo de memória, mas torna o processo de pesquisa mais lento.

Ou seja, não tem "receita de bolo"! Devemos primeiro entender bem o problema que precisamos resolver para, assim, escolher uma das dezenas de opções de árvores que temos à nossa disposição.

8.4 Ordenação

Algumas vezes precisamos que nossas árvores sejam *ordenadas*, obrigando os dados em seus nós a seguirem alguma ordenação natural, seja numérica, alfabética ou alguma outra ordenação. Qualquer árvore pode ser ordenada, caso necessário. Para isso, apenas devemos seguir duas regras, que não necessariamente devem ser aplicadas em conjunto:

- A ordenação é sempre da esquerda para a direita;
- *Filhos* à esquerda devem ser menores do que o pai e *filhos* à direta, maiores do que o pai.

Exemplos destas regras estão nas árvores da imagem a seguir:

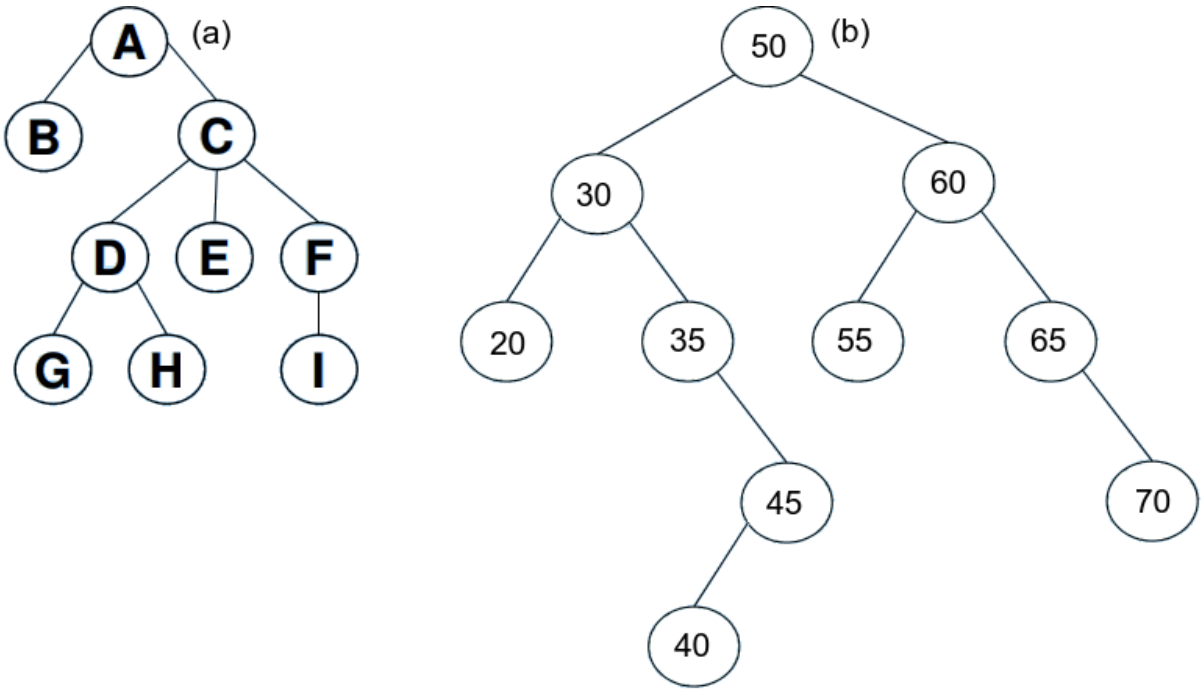


Figura 8.18: Árvores ordenadas.

Em `(a)`, temos um *árvore clássica* onde os nós estão todos ordenados entre si, da esquerda para a direita. Já em `(b)`, temos uma *árvore binária* onde os nós estão ordenados em relação a *pais* e *filhos*. Vale ressaltar que as únicas árvores que são **obrigatoriamente** ordenadas é a **Binária de busca** e **AVL**, lembrando que esta última é um tipo mais específico da primeira. As demais, só se for necessário ao problema que o uso de árvores ajuda a resolver. Por exemplo, se usarmos uma *árvore ordenada* para armazenar as matrículas de alunos de uma universidade, quando formos realizar uma pesquisa é melhor que ela esteja ordenada, pois assim garantimos que pesquisaremos em no máximo 50% das matrículas e não em todas. Isso vem do fato de que a ordenação leva a uma distribuição onde 50% dos valores estarão à

esquerda e os outros 50% à direita. Assim, ao comparar com a matrícula de referência, saberíamos para que lado ir: para matrícula menores, à esquerda, e maiores, à direita.

8.5 Implementação

Como vimos na seção *Tipos*, existem dezenas de tipos de árvores e apresentar todas seria inviável. Da mesma forma, apresentar as codificações para todas também seria impraticável. Precisaríamos de um livro focado somente em árvores para realizar estas duas tarefas. Assim, nesta seção, apenas exploraremos três tipos de árvores, que utilizarão, cada uma, uma das três formas de navegação.

Cada implementação terá seus códigos específicos. É válido ressaltar que, em cada tipo de árvore, algumas operações podem ter comportamentos diferentes devido às suas regras de funcionamento, ou mesmo algumas operações nem façam sentido existir para alguns tipos de árvores. Outra observação válida é que, embora conceitualmente somente os *pais* tenham conhecimento dos *filhos*, em determinados casos é de grande utilidade usar um ponteiro para indicar quem é o pai de um determinado nó. Isso ajuda muito nas implementações em C, evitando *dangling pointer*. Assim, em algumas implementações utilizaremos essa abordagem.

Clássica

Como vimos anteriormente, na *árvore clássica*, cada nó pode possuir uma quantidade variada de filhos. Para implementar essa característica de forma mais simples, vamos limitar a quantidade máxima de filhos de um nó em 3, mas poderíamos ter mais filhos ou, ainda, não limitar essa quantidade. Poderíamos também usar um vetor em vez de criar os ponteiros para os três filhos, mas seria também uma implementação um pouco mais árdua. Assim, para facilitar o entendimento, a estrutura de nossos nós é a seguinte:

```c
typedef struct no {
 char valor;
 struct no *pai;
 struct no *filho1;
 struct no *filho2;
 struct no *filho3;
} No;

No* criar_no(No *pai, char elemento) {

 No *no = malloc(sizeof(No));
 no->valor = elemento;
 no->pai = pai;
 no->filho1 = NULL;
 no->filho2 = NULL;
 no->filho3 = NULL;

 return no;
}
```

No primeiro código, o `struct` chamado `No` aglutina as informações que um nó da árvore pode armazenar: os ponteiros para os três filhos possíveis, o elemento pai e o valor do nó. O segundo código é responsável por criar um nó, que pode ser a raiz da árvore ou outro nó qualquer dela. Para isso, essa função recebe dois parâmetros: o elemento que o nó armazenará e seu pai — lembrando que, excepcionalmente, a raiz é o único nó que não possui um pai.

Para compreender como esse tipo de árvore pode ser manipulado, levemos em consideração a árvore a seguir:

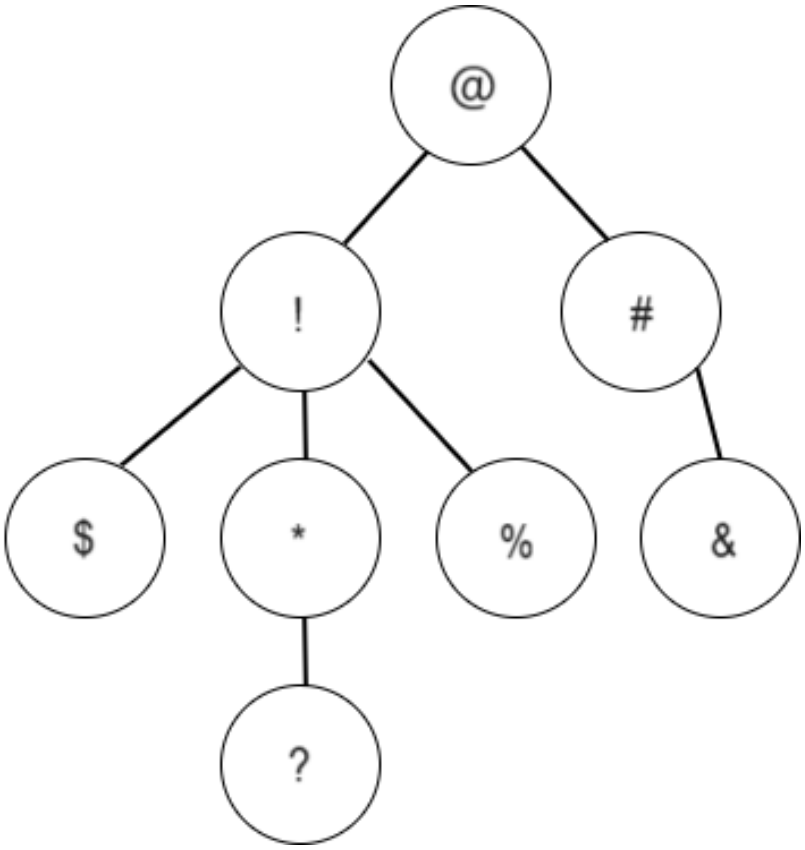


Figura 8.19: Árvore Clássica.

Nesta, vamos realizar as seguintes operações:

- Adicionar à raiz o elemento `@`;
- Adicionar os filhos da raiz `!` e `#`; os filhos `$`, `*` e `%` de `!`; o filho `&` de `#`; e o filho `?` de `*`;
- Pesquisar o elemento `*`;
- Percorrer a árvore em *pré-ordem*;
- Atualizar o valor de `!` para `+`;
- Remover o nó `*`.

Adicionando à raiz o elemento `@`

Para adicionar o valor da raiz de nossa árvore, devemos criar um nó que se comporte como tal.

```
void criar_raiz(char elemento) {/*I*/
```

```
    if (raiz == NULL) {

        raiz = criar_no(NULL, elemento);
    } else {
        printf("Raiz já foi criada\n");
    }
}

No* criar_raiz(char elemento) {

 if (raiz == NULL) { /*II*/

  No *no =  criar_no(NULL, elemento); /*III*/
  return no;
 } else {

  printf("Raiz já foi criada\n");
  return raiz; /*III*/
 }
}
```

O código anterior realiza estas operações:

I. A função recebe como parâmetro o elemento a ser adicionado como raiz;

II. Se ela não existir, cria a raiz através da função `criar_no`, onde o pai é nulo (`NULL`) e o seu valor é o `elemento` e retorna a raiz inicializada;

III. Caso a raiz já tenha sido criada anteriormente, ela é retornada.

Adicionando os filhos

Para adicionar os filhos `!` e `#` à nossa raiz e, depois, adicionar os filhos `$`, `*` e `%` no nó `!`, `?` no nó `*` e `&` no nó `#`, devemos utilizar a função `insert`. Ela é responsável por, a partir da raiz, localizar o nó que será o pai do filho que desejamos inserir. Para realizar essa operação, a função recebe três parâmetros:

- `*raiz` : a raiz da árvore, que serve de ponto de partida para se encontrar o nó a ter filhos adicionados. Essa procura utilizará o método `search`, que será explicado mais adiante;
- `pai` : valor do nó que será o pai do novo nó a ser inserido;
- `elemento` : valor do novo nó a ser inserido.

Para de fato realizar a inserção de um novo nó, o código a seguir deve ser utilizado:

```
void insert(No *raiz, char pai, char elemento) {

 if (raiz == NULL) { /*I*/
  printf("Raiz nula. Insert de nó não permitido!\n");
  return;
 }

 No *no_pai = search(raiz, pai); /*II*/

 if (no_pai != NULL) { /*III*/

  if (no_pai->filho1 == NULL) { /*IV*/
   no_pai->filho1 = criar_no(no_pai, elemento);
  } else if(no_pai->filho2 == NULL) {
   no_pai->filho2 = criar_no(no_pai, elemento);
  } else if(no_pai->filho3 == NULL) {
   no_pai->filho3 = criar_no(no_pai, elemento);
  } else { /*V*/
   printf("O nó %c não possui mais filhos disponíveis.\n", pai);
  }
 } else { /*VI*/
  printf("O nó pai %c não pertence a árvore.\n", pai);
 }
}
```

Essa função:

I. Na primeira execução, verifica se a raiz é nula. Se for, a inserção não pode ser realizada, então uma mensagem é exibida e a função é finalizada;

II. Caso não seja nula, procura-se na árvore o nó que corresponde ao possível pai informado;

III. Se o pai for encontrado, segue para os passos IV ou V. Se não, a mensagem indicando isso é exibida em VI;

IV e V. Tendo o pai sido encontrado, verifica-se qual de seus filhos está disponível para uso e, para este, é criado um novo nó que tem como pai o pai encontrado, que é o elemento passado como parâmetro. Caso nenhum filho esteja livre, uma mensagem é exibida indicando essa situação.

Para tornar mais claro, vamos descrever a inserção do elemento `*`, filho de `!`, considerando que `$` já foi inserido.

1. A raiz não é nula, pois, se estamos inserindo `*`, com certeza `@` já foi inserido. Assim, o passo I retorna `false`;
2. Como estamos inserindo `*` como filho de `!`, também sabemos que `!` já foi inserido, então o passo II retornará o nó que representa o elemento `!`;
3. Como este elemento foi encontrado, então III não é nulo, e os passos referentes a `filho1`, `filho2` ou `filho3` podem ser executados;
4. No caso, `*` é segundo filho de `!`, então a linha `no_pai->filho2 == NULL` será avaliada como `true` e `*` será inserido como segundo *filho* de `!`.

Caso todos os filhos de `!` estivessem em uso, o `else` de V seria executado. Caso passássemos como *pai* um elemento que não pertencesse à árvore, III seria nulo e VI seria executado.

Pesquisando o elemento `*`

Como dissemos no início deste capítulo, a *recursividade* é muito útil no manuseio de árvores. A função `search` utiliza esse mecanismo para tornar mais simples o processo de procura. A pesquisa sempre se inicia da *raiz* e segue navegando pelos *nós* da árvore. Ou seja, na

primeira chamada dessa função, o parâmetro *nó* é a *raiz*, mas nas seguintes serão *nós internos* ou *nós folhas*.

```
No* search(No *no, char elemento) { /*I*/

 if(no != NULL) { /*II*/

  if (no->valor == elemento) { /*III*/
   return no;
  } else {

    No *no_procurado;

    no_procurado = search(no->filho1, elemento); /*IV*/
    if (no_procurado != NULL) /*V*/
     return no_procurado; /*VI*/

    no_procurado = search(no->filho2, elemento);
    if (no_procurado != NULL)
     return no_procurado;

    no_procurado = search(no->filho3, elemento);
    if (no_procurado != NULL)
     return no_procurado;
  }
 }
 return NULL; /*VII*/
}
```

Essa função:

I. Recebe como parâmetro o nó a ser comparado com o `elemento` procurado, lembrando que o nó poderá ser a raiz ou os demais nós da árvore;

II. Se `no` for nulo (`NULL`), duas situações podem ser inferidas: ou a raiz é nula ou chegamos a um nó folha. Em ambos os casos, isso define um ponto de parada para a pesquisa e indica que o elemento não foi encontrado na árvore e, então, a linha VII é executada;

III. Não sendo nulo, é verificado se o valor do nó é igual ao elemento procurado (`no->valor == elemento`). Se for, este nó é retornado;

IV, V e VI. Caso não seja, a *recursividade* entra em ação e opera da seguinte forma: para cada *filho* do `no` corrente, vai navegando em sua *subárvore* e perguntando novamente pela igualdade (`no->valor == elemento`). Quando ele for encontrado, será retornado em VI. Caso não seja, a linha VII é executada.

Para tornar mais claro esse código, vamos descrever a procura do elemento `*` , filho de `!` :

1. Na primeira execução, `no` é a raiz e não é nula, então fazemos a verificação `no->valor == elemento` , que é `false` , pois `@` é diferente de `*` ;
2. Assim, começamos recursivamente a procurar pelos "filhos1" da árvore. Passaremos por `!` e `$` , que retornaram `false` novamente para `no->valor == elemento` ;
3. Dessa forma, começaremos a passar pelos "filhos2", lembrando que sempre navegamos da esquerda para a direita. Na segunda rodada de pesquisa, o *filho2* de `!` é justamente o `*` . Então, `no_procurado` é não nulo, o que faz com que a linha `return no_procurado;` referente a `no->filho2` seja executada. Dessa forma, `search` retorna o nó cujo elemento tem o valor igual a `*` .

É válido lembrar que, se em todas as navegações recursivas pelos filhos 1, 2 e 3 a verificação `no->valor == elemento` for `false` , a linha `return NULL;` será executada, o que indica que o elemento não pertence a nenhum nó da árvore.

Atualizando o valor de `!` para `+`

Para *atualizar um nó* de nossa árvore, usamos a função `update` . Essa é a função mais simples de nossa árvore clássica. Com o uso da função `search` , procuramos o nó compatível com o elemento a ser

substituído. Se o acharmos, alteramos seu valor. Caso contrário, uma mensagem informa que não é possível realizar a atualização.

```
void update(No *raiz, char elemento, char novo_valor) { /*I*/

 No *no = search(raiz, elemento); /*II*/

 if(no != NULL) {
  no->valor = novo_valor; /*III*/
 } else {
  printf("Nó não encontrado. Impossível realizar update."); /*IV*/
 }
}
```

Essa função:

I. Recebe como parâmetro a raiz, o elemento a ser procurado e o valor a substituí-lo;

II. Procura na árvore o nó compatível com o elemento a ser modificado;

III. Caso o nó seja encontrado, altera seu valor;

IV. Se não, uma mensagem informa que não foi possível realizar a modificação.

Embora seja uma função simples, vamos seguir o padrão e descrever a execução de como fazer a atualização de `!` para `+`.

1. Na execução da função, passamos os valores da raiz, `!` e `+`;
2. A variável `no` assumirá o valor referente ao nó da árvore cujo nó tem seu valor igual a `!`;
3. Com o nó desejado em mãos, mudamos seu valor para `+`.

É válido lembrar que se o nó compatível com o elemento desejado não for encontrado, a mensagem "Nó não encontrado. Impossível realizar update." será exibida, o que indica que o elemento que se deseja alterar não pertence à nossa árvore.

Removendo o nó *

Nesse tipo de árvore, adotamos a seguinte abordagem: a exclusão de um nó automaticamente *exclui todos os seus descendentes*. Ou seja, toda a *subárvore* enraizada pelo nó a ser excluído é removida. Veremos a seguir o código que realiza essa operação. É válido ressaltar que, para a exclusão implementada, foi feita uma pesquisa prévia para achar o nó a ser excluído.

```
void delete(No *no) { /*I*/

 No *pai = no->pai; /*II*/

 delete_subtree(no); /*III*/

 if (pai != NULL) {

  if (pai->filho1 == no) { /*IV*/
   pai->filho1 = NULL;
  }
  if (pai->filho2 == no) { /*V*/
   pai->filho2 = NULL;
  }
  if (pai->filho3 == no) { /*VI*/
   pai->filho3 = NULL;
  }
 }
}

void delete_subtree(No *no) {

 if(no != NULL) {

  delete_subtree(no->filho1);
  delete_subtree(no->filho2);
  delete_subtree(no->filho3);

  free(no);
  no = NULL;
```

```
 }
}
```

Essas funções:

I. Recebem como parâmetro o nó a ser removido;

II. Salva uma referência do pai do nó a ser excluído para evitar *dangling pointer*, o que corromperia a árvore;

III. A partir do *nó raiz* da *subárvore*, recursivamente apaga todos os seus descendentes, com o auxílio da função `delete_subtree` ;

IV, V e VI. Identifica qual dos filhos do nó pai foi excluído para colocá-lo como `NULL` .

Percorrendo a árvore em Pré-ordem

Para finalizar nossas codificações sobre árvore clássica, vamos apresentar a navegação em *pré-ordem*. Como tínhamos dito na seção *Pré-ordem (pre-order)*, essa navegação começa sua exibição sempre a partir da *raiz* e vai navegando e exibindo os valores através dos demais nós, que são os descendentes da raiz. Considerando uma navegação à esquerda, começaremos do `filho1` até o `filho3` .

```
void exibir_pre_ordem(No *no) { /*I*/

 if(no != NULL) { /*II*/

  printf("%c ",no->valor); /*III*/
  exibir_pre_ordem(no->filho1); /*IV*/
  exibir_pre_ordem(no->filho2);
  exibir_pre_ordem(no->filho3);
 }
}
```

Essa função:

I. Recebe como parâmetro para a primeira execução a raiz de uma árvore e, em posteriores chamadas recursivas, os demais nós;

II. Verifica se é nulo, ou seja, se chegamos a um nó folha ou se a raiz é nula;

III. Se não é nulo, exibe o valor do nó;

IV. Após a exibição, navega — recursivamente — em cada filho, da esquerda (`filho1`) para a direita (`filho3`), sempre exibindo seu valor inicialmente.

É válido ressaltar que, pela natureza de ser em *pré-ordem*, ou seja, exibir primeiro para depois executar a navegação, o código `printf("%c ",no->valor);` fica logo no início do `if` .

Levando em consideração nossa árvore da Figura 8.19 e após a execução dos passos anteriores, a chamada a essa função tem o seguinte resultado: `@ + $ % # &`

> Para obter o exemplo completo desta *árvore clássica* em C, onde se pode ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-ArvoreClassica.git. Lá você poderá fazer o download do código e executá-lo.

AVL

Como anteriormente visto, a *árvore AVL* é um tipo particular de *árvore binária* onde, além da limitação de dois *nós filhos*, existe a preocupação com a diferença de alturas entre *subárvores*. Além disso, vimos que esta é obrigatoriamente ordenada e, por isso, mudaremos os valores armazenados de `char` para `int` para explicitarmos a ordenação. Para implementar esse tipo de árvore, podemos iniciar com a apresentação dos dois códigos a seguir:

```
typedef struct no {
 int valor;
```

```
 struct no *filho_esquerdo;
 struct no *filho_direito;
 int altura;
} No;

No* criar_no(int elemento) {

 No *no = malloc(sizeof(No));
 no->valor = elemento;
 no->filho_esquerdo = NULL;
 no->filho_direito = NULL;
 no->altura = 0;

 return no;
}
```

No primeiro código, o `struct` chamado `No` agora possui apenas dois filhos: o esquerdo e o direito. Além disso, temos a variável para guardar o valor do nó (`valor`) e uma novidade: a variável `altura` . Ela contribui na verificação da diferença entre as profundidades das subárvores e, portanto, para detectar se rotações precisam ser realizadas para manter o balanceamento da árvore. Novamente, o segundo código é responsável por criar um nó, que pode ser a raiz da árvore ou qualquer outro nó dela. Nessa árvore, a função recebe só um parâmetro: o elemento que o nó armazenará. Por fim, a `altura` é iniciada com `0` , pois todo novo nó é *folha*, ou seja, não possui *subárvores* à esquerda ou à direita.

Para facilitar o entendimento de como de nossa árvore AVL pode ser manipulada, levemos em consideração a árvore a seguir:

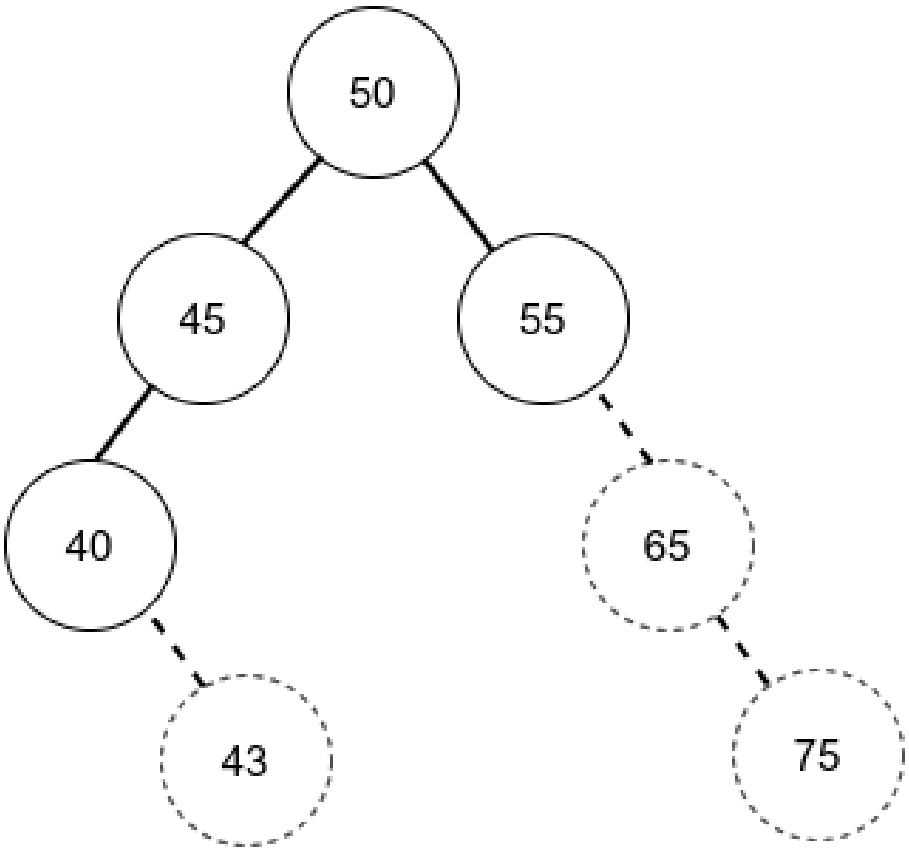


Figura 8.20: Árvore AVL.

Vamos realizar as seguintes operações:

- Adicionar à raiz o elemento `50`;
- Adicionar o filho esquerdo `45` e o direito `55` da raiz;
- Adicionar o filho esquerdo `40` de `45`;
- Adicionar o filho direito `43` de `40`, para gerar uma rotação dupla à esquerda;
- Adicionar o filho direito `65` de `55` e depois o filho direito `75` de `65`, para uma rotação simples à esquerda;
- Pesquisar o elemento `40`;
- Remover o nó `65`;
- Percorrer a árvore *em-ordem*.

Antes de apresentarmos as operações, é importante frisar que nesse tipo de árvore algumas funções auxiliares devem ser utilizadas, que são apresentadas e explicadas a seguir.

```
int maior_altura(int altura_esquerda, int altura_direita) {

 if(altura_esquerda > altura_direita) {
  return altura_esquerda;
 } else {
  return altura_direita;
 }
}
```

A função `maior_altura` é responsável por identificar qual *subárvore* é maior, ou seja, qual causa o desbalanceamento.

```
int altura_no(No *no) {

 if (no == NULL) {
  return -1;
 } else {
  return no->altura;
 }
}
```

A função `altura_no` é responsável por identificar a *altura* relativa de um determinado nó dentro da árvore.

```
int valor_balanceamento(No *no) {

 if(no != NULL) {
  return altura_no(no->filho_esquerdo) - altura_no(no->filho_direito);
 } else {
  return 0;
 }
}
```

A função `valor_balanceamento` é responsável por identificar para que lado a árvore está desbalanceada. Lembremos que a lista de valores que indica que tipo de desbalanceamento a árvore possui foi apresentada no início da seção *AVL*.

```
No* balancear(No *no) {
```

```
 int balanceamento = valor_balanceamento(no); /*I*/

 if(balanceamento < -1 && valor_balanceamento(no->filho_direito)  <= 0) {
/*II*/
  no = rotacao_esquerda(no);
 } else if (balanceamento > 1 && valor_balanceamento(no->filho_esquerdo)
>= 0) { /*III*/
  no = rotacao_direita(no);
 } else if (balanceamento > 1 && valor_balanceamento(no->filho_esquerdo)
< 0) { /*IV*/
  no = rotacao_dupla_esquerda(no);
 } else if (balanceamento < -1 && valor_balanceamento(no->filho_direito)
> 0) { /*V*/
  no = rotacao_dupla_direita(no);
 }

 return no;
}
```

A função `balancear` é responsável por realizar as rotações para que a árvore se torne balanceada. Tais rotações são realizadas a partir do desbalanceamento de um determinado nó. Para isso, o código anterior realiza os seguintes passos:

I. Calcula o *balanceamento* de um determinado nó que se comporta como a raiz de uma subárvore no momento do balanceamento;

II, III, IV e V. Novamente, com base na lista de valores que indicam os desbalanceamentos e no valor de balanceamento de cada filho da raiz atualmente analisada, realizamos a rotação adequada.

```
No* rotacao_esquerda(No *no) {

 No *subarvore_D, *subarvore_E_D; /*I*/

 subarvore_D = no->filho_direito; /*II*/

 subarvore_E_D = subarvore_D->filho_esquerdo; /*III*/
 subarvore_D->filho_esquerdo = no; /*IV*/
```

```
 no->filho_direito = subarvore_E_D; /*V*/

 no->altura = maior_altura(altura_no(no->filho_esquerdo), altura_no(no-
>filho_direito)) + 1; /*VI*/
 subarvore_D->altura = maior_altura(altura_no(subarvore_D-
>filho_esquerdo), altura_no(subarvore_D->filho_direito)) + 1; /*VII*/

 return subarvore_D;
}
```

A função `rotacao_esquerda` é responsável por implementar o algoritmo da seção *Rotação simples à esquerda* em um determinado nó. Em I, inicialmente são criadas variáveis que auxiliarão na execução do algoritmo: `subarvore_D` , que representa a *subárvore à direita*, e `subarvore_E_D` , que representa a *subárvore à esquerda* da subárvore à direita separada. Dessa forma, temos:

- O passo `1` do algoritmo é executado por II;
- O passo `2` do algoritmo é executado por III e IV;
- O passo `3` do algoritmo é executado por V.

Por fim, como o nó corrente que representa a raiz de uma determinada subárvore foi mudado de lugar, VI recalcula sua altura relativa dentro da árvore e VII recalcula a altura relativa da *subárvore à direita*, que também foi movimentada.

```
No* rotacao_direita(No *no) {

 No *subarvore_E, *subarvore_D_E; /*I*/


 subarvore_E = no->filho_esquerdo; /*II*/

 subarvore_D_E = subarvore_E->filho_direito; /*III*/
 subarvore_E->filho_direito = no; /*IV*/

 no->filho_esquerdo = subarvore_D_E; /*V*/
```

```
 no->altura = maior_altura(altura_no(no->filho_esquerdo), altura_no(no-
>filho_direito)) + 1; /*VI*/
 subarvore_E->altura = maior_altura(altura_no(subarvore_E-
>filho_esquerdo), altura_no(subarvore_E->filho_direito)) + 1; /*VII*/

 return subarvore_E;
}
```

A função `rotacao_direita` é responsável por implementar o algoritmo da seção *Rotação simples à direita* em um determinado nó. Em I, inicialmente são criadas variáveis que auxiliarão na execução do algoritmo: `subarvore_E` , que representa a subárvore à esquerda, e `subarvore_D_E` , que representa a *subárvore à direita* da subárvore à esquerda separada. Dessa forma, temos:

- O passo `1` do algoritmo é executado por II;
- O passo `2` do algoritmo é executado por III e IV;
- O passo `3` do algoritmo é executado por V.

Por fim, como o nó corrente que representa a raiz de uma determinada subárvore foi mudado de lugar, VI recalcula sua altura relativa dentro da árvore e VII recalcula a altura relativa da *subárvore à esquerda*, que também foi movimentada.

```
No* rotacao_dupla_esquerda(No *no) {

 no->filho_esquerdo = rotacao_esquerda(no->filho_esquerdo); /*I*/
 return rotacao_direita(no); /*II*/
}
```

A função `rotacao_dupla_esquerda` é responsável por implementar o algoritmo da seção *Rotação dupla à esquerda* em um determinado nó. Ao contrário da *rotação simples à esquerda*, a *dupla à esquerda* possui uma codificação muito fácil, pois é basicamente o reúso das implementações das rotações simples: *à direta* e *à esquerda*. Dessa forma, temos:

- O passo `1` do algoritmo é executado por I;
- O passo `2` do algoritmo é executado por II.

O recálculo das alturas é feito internamente pelas funções reusadas.

```
No* rotacao_dupla_direita(No *no) {

 no->filho_esquerdo = rotacao_direita(no->filho_direito); /*I*/
 return rotacao_esquerda(no); /*II*/
}
```

A função `rotacao_dupla_direita` é responsável por implementar o algoritmo da seção *Rotação dupla à direita* em um determinado nó. Ao contrário da *rotação simples à direita*, a *dupla à direita* possui uma codificação muito fácil, pois é basicamente o reúso das implementações das rotações simples: *à esquerda* e *à direta*. Dessa forma, temos:

- O passo `1` do algoritmo é executado por I;
- O passo `2` do algoritmo é executado por II.

O recálculo das alturas são feitos internamente pelas funções reusadas.

Finalmente, ao fim da apresentação destas funções auxiliares, podemos começar de fato a manipular nossa árvore AVL.

Adicionando a raiz `50`

Para adicionar o valor da raiz de nossa árvore, devemos criar um nó que se comporte como tal.

```
void criar_raiz(char elemento) { /*I*/

  if (raiz == NULL) {
   raiz = criar_no(elemento); /*II*/
  } else {
   printf("Raiz já foi criada\n"); /*III*/
  }
}
```

Explicando as operações do código anterior, em:

I. A função recebe como parâmetro o elemento a ser adicionado como raiz;

II. Se ela não existir, cria a raiz através da função `criar_no` ;

III. Caso a raiz já tenha sido criada, tal fato é informado.

Adicionando os filhos

A função `insert` é responsável por, a partir da raiz, localizar o ponto onde a inserção deve ser feita, que é sempre o ponto nulo (`NULL`) à esquerda ou à direita de um nó folha. Assim, a usamos para adicionar:

- O filho esquerdo `45` e o direito `55` à nossa raiz `50` ;
- O filho esquerdo `40` de `45` e o filho direito `43` de `40` ;
- O filho direito `65` de `55` ;
- O filho direito `75` de `65` .

Ao contrário da árvore clássica, na AVL não passamos o *pai*, pois não podemos escolhê-lo. Na verdade, por ser ordenada, o novo nó terá um local específico dentro da árvore, que será encontrado após uma pesquisa recursiva. Para realizar essa operação, a função recebe dois parâmetros:

- `*no` : nó que serve de ponto de partida para se encontrar a posição onde o novo filho deve ser adicionado. Na primeira execução, `no` é a raiz da árvore. Nas demais execuções, serão os outros nós da árvore;
- `elemento` : valor do novo nó a ser inserido.

Assim, para de fato realizar a inserção de um novo nó, o código a seguir deve ser utilizado:

```
No* insert(No *no, int elemento) { /*I*/

 if (no == NULL) {
  return criar_no(elemento); /*II*/
 } else {
```

```
  if(elemento < no->valor) { /*III*/
   no->filho_esquerdo = insert(no->filho_esquerdo, elemento);
  } else if(elemento > no->valor) { /*IV*/
   no->filho_direito = insert(no->filho_direito, elemento);
  } else { /*V*/
   printf("INSERT não realizado, pois elemento já pertence a árvore");
  }
 }

 no->altura = maior_altura(altura_no(no->filho_esquerdo),altura_no(no-
>filho_direito)) + 1; /*VI*/

 no = balancear(no); /*VII*/

 return no; /*VIII*/
}
```

O código anterior:

I. Recebe como parâmetro o elemento a ser adicionado e o nó para auxiliar na localização do ponto de inserção. Na primeira execução, `no` é a raiz e, nas subsequentes, os demais *nós* da árvore;

II. Se `no` for nulo, é sinal que achou o local de inserção, então um novo nó é criado e retornado;

III, IV e V. Nestes três passos, a AVL realmente se materializa. Recursivamente, navega-se para a esquerda se o elemento a ser incluído for menor que seu pai; ou para a direita se o elemento for maior que seu pai. Caso não seja nenhuma dessas opções, é sinal de que o elemento já existe na árvore e ele não será inserido;

VI. Independentemente de a inserção ter ocorrido ou não, calcula-se a altura relativa do nó manipulado atualmente dentro da árvore. Se uma inserção ocorreu de fato, seu valor mudará; se não, apenas será atualizado para o mesmo valor;

VII. A altura do nó calculado no passo anterior influenciará na execução ou não do balanceamento através da função `balancear`;

VIII. Ao final de todo o desempilhamento das chamadas recursivas para determinar o local de inserção, `no` representará a raiz da árvore, que será retornada. É válido ressaltar que ela pode ter mudado a depender de possíveis rotações executadas em decorrência de alguma inserção.

Pesquisando o elemento `40`

A pesquisa em uma árvore AVL é muito semelhante à de uma árvore clássica. Chamadas recursivas são realizadas e verificações de nulo (`NULL`) e de igualdade (`no->valor == elemento`) são feitas para determinar o ponto de parada. A diferença está em como a recursividade é acionada. No caso, verifica-se se o valor é menor ou maior do que o do nó corrente, para, assim, navegar para a esquerda ou para a direita, respectivamente.

```
No* search(No *no, int elemento) { /*I*/

 if (no == NULL) { /*II*/
  return NULL;
 } else if(no->valor == elemento) { /*III*/
  return no;
 } else if(elemento < no->valor) { /*IV*/
  search(no->filho_esquerdo, elemento);
 } else {
  search(no->filho_direito, elemento);
 }
}
```

A função anterior:

I. Recebe inicialmente a raiz da árvore e, posteriormente, os demais nós. Além disso, recebe o elemento a ser encontrado;

II. Se chegarmos a um nó nulo, é sinal de que percorremos toda a árvore e não encontramos o nó que desejávamos;

III. Se em algum momento o valor do nó for igual ao elemento de pesquisa (`no->valor == elemento`), então o nó foi encontrado dentro da árvore e deve ser retornado;

IV. Neste passo, novamente a AVL se materializa. Caso o elemento seja *menor* que o valor do nó (`elemento < no->valor`), navegamos recursivamente para as subárvores *à esquerda*. Se for *maior*, navegamos recursivamente para as subárvores *à direita*. A execução desse passo dispara as execuções de II, III e do próprio IV.

Removendo o nó 65

Na remoção de um nó em uma árvore AVL, apenas o nó em questão é excluído. Como consequência desse fato, rotações podem ser realizadas para manter o balanceamento e a ordenação da árvore. Para isso, o código a seguir deve ser executado.

```
No* delete(No *no, int elemento) {/*I*/

 if(no == NULL) { /*II*/
  return NULL;
 } else if (no->valor != elemento) { /*III*/

  if(elemento < no->valor) {
   no->filho_esquerdo = delete(no->filho_esquerdo, elemento);
  } else {
   no->filho_direito = delete(no->filho_direito, elemento);
  }
 } else {

  if(folha(no)) { /*IV*/

   free(no);
   no = NULL;

   return NULL;
  } else if(no->filho_esquerdo != NULL && no->filho_direito != NULL) {
/*V*/
```

```
   No *no_filho = no->filho_direito;

   while(no_filho->filho_direito != NULL) {
    no_filho = no_filho->filho_direito;
   }

   no->valor = no_filho->valor;
   no_filho->valor = elemento;
   no->filho_direito = delete(no->filho_direito, elemento);

   return no;
  } else { /*VI*/

   No *no_filho;

   if(no->filho_esquerdo != NULL) {
    no_filho = no->filho_esquerdo;
   } else {
    no_filho = no->filho_direito;
   }

   free(no);
   no = NULL;

   return no_filho;
  }
 }

 no->altura = maior_altura(altura_no(no->filho_esquerdo),altura_no(no-
>filho_direito)) + 1; /*VII*/

 no = balancear(no); /*VIII*/

 return no; /*IX*/
}
```

A função anterior:

I. Recebe como parâmetro um *nó* e o elemento a ser excluído. Na primeira execução, o nó é a raiz e, nas demais, os outros nós da árvore;

II. Se `no` chegou a nulo (`NULL`) em algum momento, é sinal de que o elemento não foi encontrado na árvore; então, a exclusão não é realizada;

III. Se não for nulo, então verifica se o valor do nó corrente é diferente do elemento (`no->valor != elemento`) a ser excluído. Se for, navega-se recursivamente pela esquerda (se valor menor) ou pela direita (se valor maior) para procurar o nó desejado;

IV. Entretanto, se for igual, verifica se é um *nó folha*, para simplesmente excluí-lo;

V. Se não for um folha, faz a substituição entre um dos filhos à esquerda ou à direita com o nó a ser excluído, caso ambos existam. Tal escolha é aleatória e, no nosso caso, foi escolhido o lado direito;

VI. Se somente um dos filhos estiver presente, faz a substituição dele (à esquerda ou à direita) com o nó a ser excluído;

VII. Independentemente de a exclusão ter ocorrido ou não, calcula-se a altura relativa do nó manipulado atualmente dentro da árvore. Se uma exclusão ocorreu de fato, seu valor mudará; se não, apenas será atualizado para o mesmo valor;

VIII. O valor da altura do nó calculado no passo anterior influenciará na execução ou não do balanceamento através da função `balancear` ;

IX. Ao final de todo o desempilhamento das chamadas recursivas para reorganizar a árvore após a exclusão, `no` representará a raiz da árvore, que será retornada. É válido ressaltar que ela pode ter mudado a depender de possíveis rotações executadas em decorrência de alguma exclusão.

Percorrendo a árvore Em-ordem

Finalizando nossas codificações para a árvore AVL, vamos apresentar a navegação *em-ordem*. Como tínhamos dito na seção *Em-ordem (in-order)*, essa navegação começa sua exibição sempre a partir do *nó folha* mais profundo, geralmente da *subárvore à esquerda*. Após toda ela ser exibida, a raiz é então exibida e, posteriormente, toda a subárvore à direita, novamente a partir do nó mais profundo.

```
void exibir_em_ordem(No *no) { /*I*/

 if(no != NULL) { /*II*/
  exibir_em_ordem(no->filho_esquerdo); /*III*/
  printf("%d ",no->valor); /*IV*/
  exibir_em_ordem(no->filho_direito); /*V*/
 }
}
```

Essa função:

I. Recebe como parâmetro para a primeira execução a raiz de uma árvore e, posteriormente, os demais nós;

II. Verifica se é nulo, ou seja, se chegamos a um nó folha ou se a raiz é nula;

III. Se não é nulo, inicia a navegação até chegar ao nó folha mais profundo à esquerda;

IV. No desempilhamento, vai exibindo todos os nós mais à esquerda e, ao final, a raiz;

V. Inicia a navegação até chegar ao nó folha mais profundo à direita e, no desempilhamento, vai exibindo todos os nós mais à direita.

É válido ressaltar que, pela natureza de ser *em-ordem*, ou seja, exibir primeiro um lado da árvore, depois a raiz e, por fim, o outro lado da árvore, o código `printf("%c ",no->valor);` fica entre o filho esquerdo e o direito da AVL.

Levando em consideração a árvore da Figura 8.20 e após a execução dos passos anteriores e de suas rotações, a chamada a essa função tem o seguinte resultado: `40 43 45 50 55 75`

> Para obter o exemplo completo desta *árvore AVL* em C, onde se pode ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-ArvoreAVL.git. Lá você poderá fazer o download do código e executá-lo.

N-Ária

Como visto previamente, na *árvore N-ária* não existe uma limitação na quantidade de filhos. Entretanto, existem pequenas peculiaridades em sua implementação: só o primeiro filho (primogênito) se relaciona com o pai; os demais são irmãos do primogênito. Para implementar estas características de forma simples, vamos utilizar o conceito de *lista*, estudado no capítulo 6. Dessa forma, as estruturas necessárias para representar nossos nós são as seguintes:

```
typedef struct no {
 char valor;
 struct no *primogenito;
 struct no *irmao;
} No;

No* criar_no(char elemento) {

 No *no = malloc(sizeof(No));
 no->valor = elemento;
 no->primogenito = NULL;
 no->irmao = NULL;

 return no;
}
```

No primeiro código, o `struct` chamado `No` aglutina as informações que um nó da árvore pode armazenar: os ponteiros para o `primogênito` e para o `irmão`, além do valor do nó. Vale ressaltar que `irmão` termina por se comportar como o `proximo` da ED *lista*. O segundo código é responsável por criar um nó, que pode ser a raiz da árvore ou um outro nó qualquer dela. Para isso, a função recebe apenas um parâmetro: o elemento que o nó armazenará.

Para compreender como esse tipo de árvore pode ser manipulado, levemos em consideração a árvore a seguir:

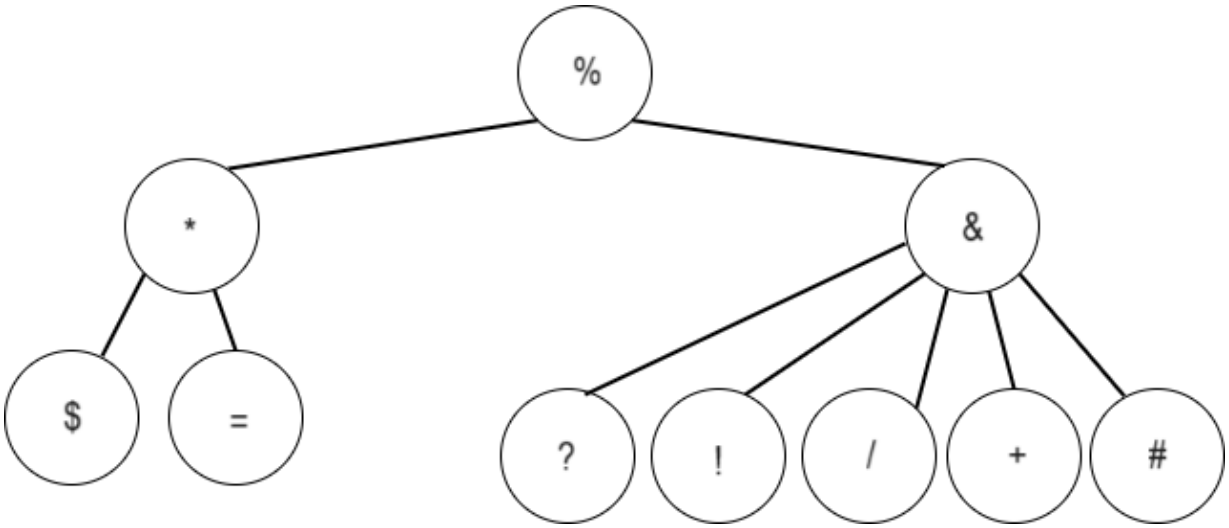


Figura 8.21: Árvore N-ária.

Nesta, vamos realizar as seguintes operações:

- Adicionar à raiz o elemento `%`;
- Adicionar o primogênito `*` da raiz e depois seu irmão `&`;
- Adicionar o primogênito `$` de `*` e depois seu irmão `=`;
- Adicionar o primogênito `?` de `&` e depois os irmãos `!`, `/`, `+` e `#`;
- Pesquisar o elemento `=`
- Atualizar o valor de `&` para `@`;
- Percorrer a árvore em Pós-ordem.

Aqui podemos ressaltar que a remoção não será implementada para esse tipo de árvore. Como essa árvore tem uma configuração bem peculiar, sua exclusão não é padronizada ou intuitiva como na árvore AVL e na clássica, respectivamente. Na *N-ária*, a exclusão depende muito do problema que está sendo solucionado, por exemplo. Devido a isso, qualquer implementação que sugeríssemos seria extremamente específica e não reusável. Além disso, a complexidade de exclusões nesse tipo de árvore também é maior do que nas outras apresentadas.

Outro fator que dificulta a implementação de uma exclusão é a forma como decidimos explorar a *N-ária*: uma árvore não ordenada. Caso tivéssemos escolhido essa abordagem, a exclusão seria mais "fácil" de implementar, mas ainda seria uma codificação árdua. Entretanto, os processos de inclusão e pesquisa também seriam penalizados e se tornariam mais complexos em relação aos aqui apresentados.

Dessa forma, para não disponibilizar um código extremamente extenso e complexo, optamos por não implementar essa exclusão e não trabalhar com uma *N-ária* ordenada. Trocamos "uma codificação mais difícil" por "duas mais fáceis", ao invés de termos "três difíceis". Mas tenhamos em mente que é possível manipulá-la destas maneiras, mas isso tem que ser bem alinhado com o problema que está se desejando resolver com o uso desse tipo de árvore.

Adicionando a raiz `%`

Para adicionar o valor da raiz de nossa árvore N-ária, assim como nas outras árvores, devemos criar um nó que se comporte como tal.

```
void criar_raiz(char elemento) { /*I*/

  if (raiz == NULL) {
   raiz = criar_no(elemento); /*II*/
  } else {
   printf("Raiz já foi criada\n"); /*III*/
```

```
  }
}
```

Explicando o código anterior:

I. A função recebe como parâmetro o elemento a ser adicionado como raiz;

II. Se ela não existir, cria a raiz através da função `criar_no` ;

III. Caso a raiz já tenha sido criada, tal fato é informado.

Adicionando os primogênitos e irmãos

Para inserir nós que não sejam a raiz, temos outras duas opções de filhos: um primogênito ou um irmão. A função `insert` é responsável por realizar essa operação. Assim, para adicionar o primogênito `*` da raiz e depois seu irmão `&` , o primogênito `$` de `*` e depois seu irmão `=` e, por fim, o primogênito `?` de `&` e depois os irmãos `!` , `/` , `+` e `#` , devemos utilizar a função:

```
void insert(No *raiz, char elemento, char pai) { /*I*/

 No *no_pai = search(raiz, pai); /*II*/

 if (no_pai != NULL) { /*III*/

  if (no_pai->primogenito == NULL) { /*IV*/

   no_pai->primogenito = criar_no(elemento);
   printf("Primogênito %c de %c inserido.\n", elemento, pai);
  } else {

   No *filho = no_pai->primogenito; /*V*/

   while(filho->irmao != NULL) {
    filho = filho->irmao;
   }

   filho->irmao = criar_no(elemento);
```

```
   printf("Filho %c de %c e irmão de %c inserido.\n", elemento, pai,
filho->valor);
  }
 } else {
  printf("Pai não encontrado. INSERT não realizado.\n"); /*VI*/
 }
}
```

O código anterior:

I. Recebe como parâmetro a raiz da árvore, o elemento a ser inserido e o pai para esse elemento;

II. Procura na árvore, a partir de sua raiz, o pai desejado;

III. Se o pai for encontrado (`no_pai != NULL`), então a inserção poderá ser realizada;

IV. Se o `primogenito` do pai encontrado for nulo (`NULL`), então estamos inserindo o primeiro filho. Assim, esse `primogenito` recebe o novo nó (`no_pai->primogenito = criar_no(elemento);`) e a mensagem que indica essa operação é exibida;

V. Caso não seja nulo, então estamos inserindo irmãos. Então, a partir do primeiro filho (`No *filho = no_pai->primogenito;`), navegamos por todos os irmãos até chegar a nulo (`while(filho->irmao != NULL)`), que indicará que o novo nó pode ser inserido ao fim da lista de irmãos. Assim, o código `filho->irmao = criar_no(elemento);` é executado;

VI. Caso um pai não tenha sido encontrado em III, uma mensagem indicativa é exibida.

Pesquisando o elemento `=`

Tanto no `insert` como no `update` , pesquisamos de forma prévia um determinado nó. Para isso, é usada a função `search` . Como exemplo, vamos pesquisar o nó `=` .

```
No* search(No *no, char elemento) { /*I*/

 if (no != NULL) { /*II*/

  if (no->valor == elemento) {
   return no;
  } else {

   No *filho = no->primogenito; /*III*/

   while(filho != NULL) {

    No *no_pesquisa = search(filho, elemento);

    if(no_pesquisa != NULL) {
     return no_pesquisa;
    }

    filho = filho->irmao;
   }
  }
 }

 return NULL; /*IV*/
}
```

Essa função:

I. Recebe como parâmetro o nó raiz na primeira execução e o elemento a ser pesquisado;

II. Se esse nó, raiz ou demais nós nas chamadas recursivas subsequentes não for nulo (`NULL`), então verificamos se o nó corrente (*irmão*, *primogênito*, *pai* ou *raiz*) possui seu valor igual ao do elemento desejado (`no->valor == elemento`). Se sim, retornamos esse nó; se não, iniciamos o processo de pesquisa recursiva;

III. Inicia pelo *primogênito* (`No *filho = no->primogenito;`) e, recursivamente (`No *no_pesquisa = search(filho, elemento);`), navega

na árvore enquanto não atingir nulo (`while(filho != NULL)`). Se em um determinado momento encontramos o nó, ele é retornado (`if(no_pesquisa != NULL)` e `return no_pesquisa;`). Se não, ficamos constantemente navegando pelos irmãos (`filho = filho->irmao`);

IV. Se chegarmos a executar este passo, é sinal de que todas as chamadas recursivas `if(no_pesquisa != NULL)` sempre retornaram `NULL` , o que fez com que, em um determinado momento, `if (no != NULL)` fosse `false` . Isso implica que toda a árvore foi percorrida e nenhum de seus nós teve o valor igual ao elemento desejado.

Atualizando o valor de `&` para `@`

Para *atualizar* um nó de nossa árvore, usamos a função `update` . Essa é a função mais simples de nossa árvore N-ária. Com o uso da função `search` , procuramos o nó compatível com o elemento a ser substituído. Se o acharmos, alteramos seu valor; caso contrário, uma mensagem informa que não é possível realizar a atualização.

```
void update(No *raiz, char elemento, char novo_valor) { /*I*/

 No *no = search(raiz, elemento); /*II*/

 if(no != NULL) { /*III*/
  no->valor = novo_valor;
 } else {
  printf("Nó não encontrado. Impossível realizar UPDATE.");
 }
}
```

Essa função:

I. Recebe como parâmetro a raiz, o elemento a ser procurado e o valor a substituí-lo;

II. Procura na árvore o nó compatível com o elemento a ser modificado;

III. Caso o nó seja encontrado, altera o seu valor.

É válido lembrar que se o nó compatível com o elemento desejado não for encontrado, a mensagem "Nó não encontrado. Impossível realizar update." será exibida, o que indica que o elemento que se deseja alterar não pertence à nossa árvore.

Percorrendo a árvore em Pós-ordem

Finalizando nossas codificações para a árvore N-ária, vamos apresentar a navegação pós-ordem. Como tínhamos dito na seção *Pós-ordem (post-order)*, essa navegação começa sua exibição sempre a partir do nó folha mais profundo, geralmente da subárvore à esquerda. Após toda esta ser exibida, toda a subárvore à direita é exibida também a partir do nó folha mais profundo. Por fim, a raiz é exibida.

```
void exibir_pos_ordem(No *no) { /*I*/

 if(no != NULL) { /*II*/

  No *no_auxiliar = no->primogenito; /*III*/

  while(no_auxiliar != NULL) { /*IV*/

   exibir_pos_ordem(no_auxiliar);
   no_auxiliar = no_auxiliar->irmao;
  }
  printf("%c ",no->valor); /*V*/
 }
}
```

Essa função:

I. Recebe como parâmetro para a primeira execução a raiz de uma árvore;

II. Verifica se é nulo, ou seja, se chegamos a um nó folha ou se a raiz é nula;

III. Se não é nulo, inicia a navegação a partir do *primogênito*, que é o mais à esquerda;

IV. Navega-se recursivamente até o irmão mais profundo à esquerda (`exibir_pos_ordem(no_auxiliar);`), sempre passando de irmão em irmão (`no_auxiliar = no_auxiliar->irmao;` . Em um determinado momento, `no_auxiliar` será nulo, e isso indicará que as chamadas recursivas chegaram a um nó filho mais profundo, seja um irmão à esquerda ou à direita;

V. Assim, o desempilhamento se iniciará e o valor da cada nó (*irmão*, *primogênito*, *pai* ou *raiz*) será exibido.

É válido ressaltar que, pela natureza de ser *pós-ordem*, ou seja, exibir por último a raiz, o código `printf("%c ",no->valor);` fica ao final da função.

Levando em consideração a árvore da Figura 8.21 e após a execução dos passos anteriores, a chamada a essa função tem o seguinte resultado: `$ = * ? ! / + # & %`

> Para obter o exemplo completo desta *árvore N-ária* em C, onde se pode ver o funcionamento real dela, acesse o link: https://github.com/thiagoleiteecarvalho/ed-ArvoreNaria.git. Lá você poderá fazer o download do código e executá-lo.

8.6 Exemplos de uso

Assim como as estruturas anteriormente apresentadas, a árvore também possui uma grande gama de aplicabilidades dentro da computação e no nosso dia a dia. Vejamos alguns exemplos a seguir.

Organização de pastas/arquivos em sistemas operacionais

O exemplo mais claro e usual de árvores é a organização das pastas e arquivos em um sistema operacional. Quando acessamos o Windows Explorer no Windows ou o Nautilus no Ubuntu, podemos navegar pelas pastas e arquivos através desses programas. O modo como as pastas e arquivos são organizados e exibidos nada mais é que a ED árvore. Basta verificarmos que a *raiz* é o diretório-base do SO — `C:` no Windows e `/` no Ubuntu — e, a partir dele, temos pastas, que são *nós internos*, e arquivos ou pastas vazias, que são *nós folhas*. Para ilustrar isso em ambos, temos um exemplo para a seguinte organização:

- Ubuntu: Dentro de `/` temos o *arquivo 1*, o *arquivo 2* e a *Pasta 1*. Dentro dessa pasta, temos a *subpasta 1* e a *subpasta 2*;
- Windows: Dentro de `C:` temos a pasta *Pasta 1* e *Pasta 2*. Dentro da primeira pasta, temos a *subpasta 1* e o *arquivo 1*; na segunda, o *arquivo 2* e o *arquivo 3*.

Para o Ubuntu, temos a imagem:

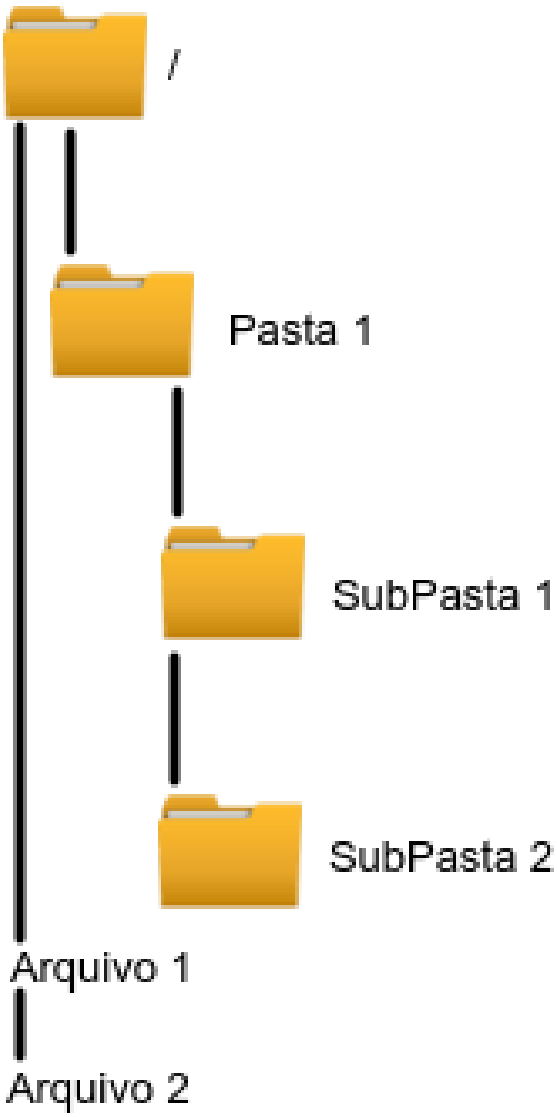


Figura 8.22: Ubuntu.

A árvore para essa organização está a seguir:

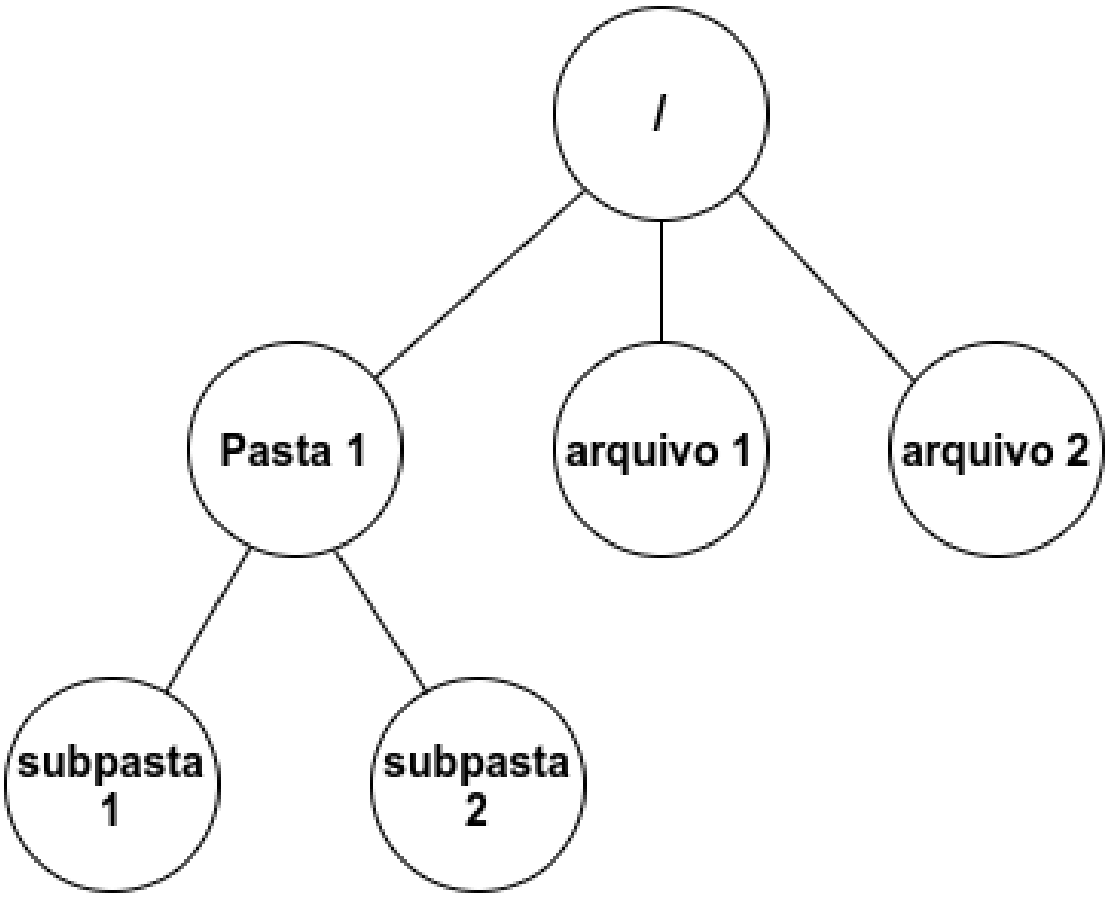


Figura 8.23: Árvore para o Ubuntu.

Para o Windows Explorer, temos a imagem:

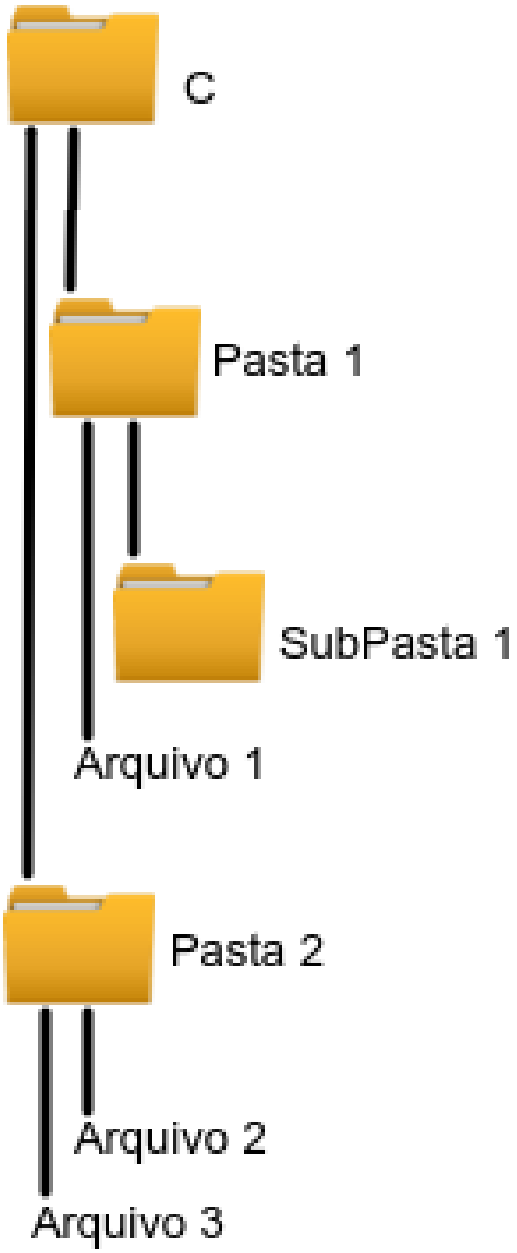


Figura 8.24: Windows.

A árvore para essa organização está a seguir:

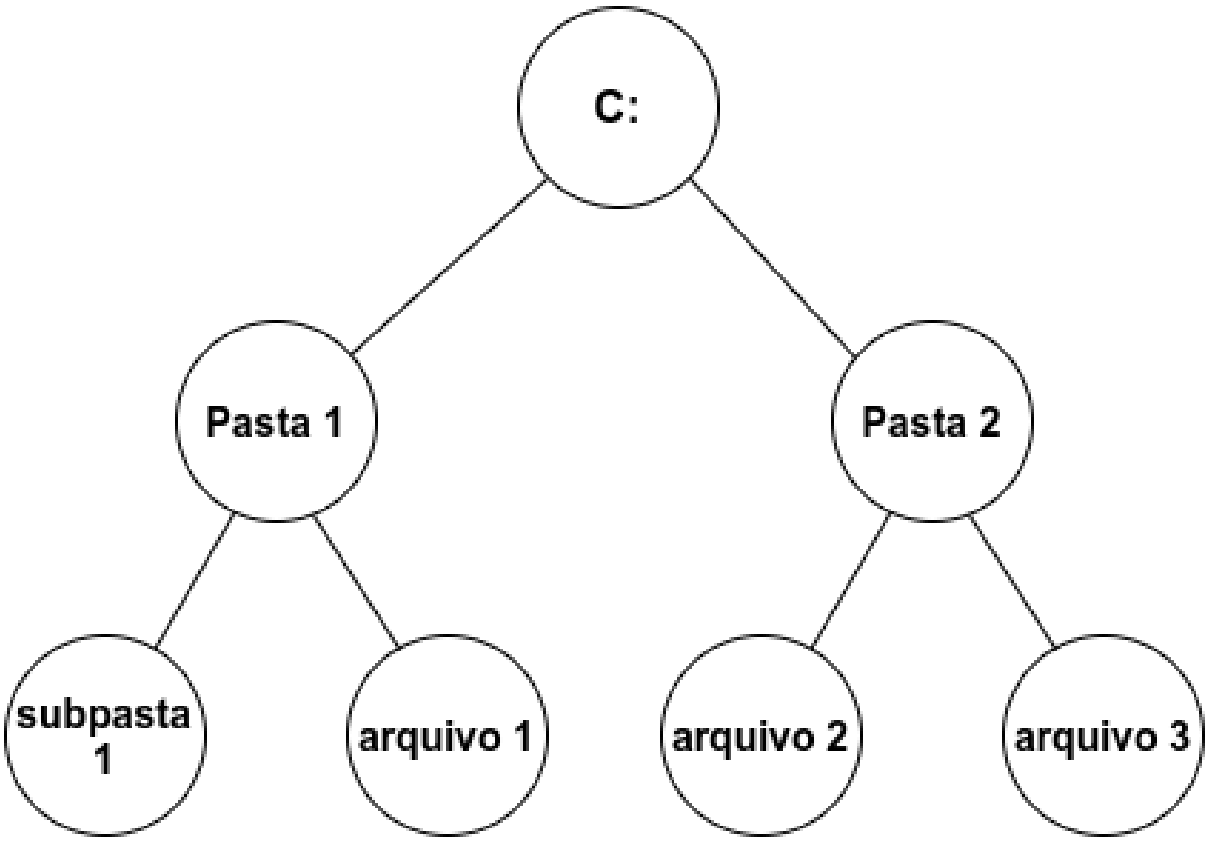


Figura 8.25: Árvore para o Windows.

Validação de expressões

No capítulo relativo a *pilha*, vimos que ela pode ser usada para verificar se expressões estão balanceadas. Podemos também usar a árvore para executar/validar uma expressão. Tendo como exemplo a expressão `(2 x (a - 1) + (3 x b))`, os *nós internos* são os operadores e os *nós externos*, os operandos. Se em algum momento na criação da árvore as duas condições não forem atendidas, a expressão é inválida.

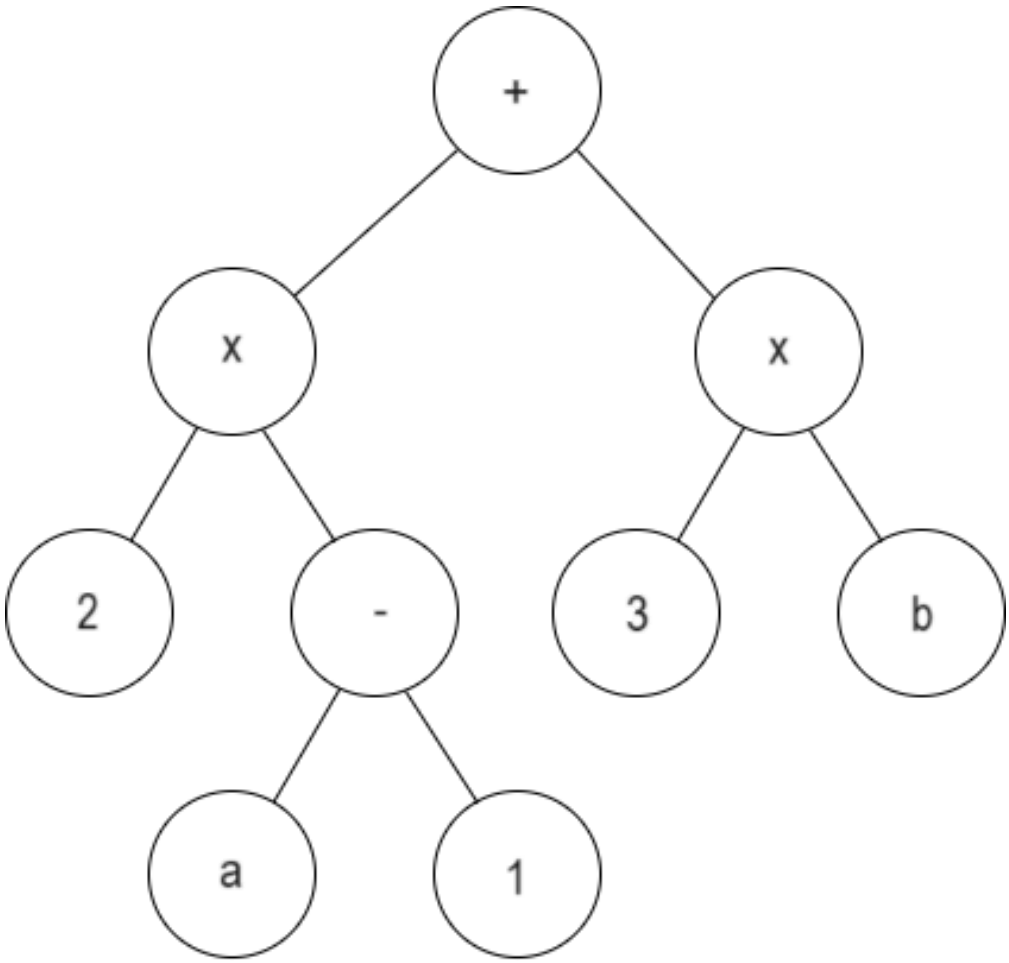


Figura 8.26: Árvore para a expressão (2 x (a - 1) + (3 x b)).

Processo de decisão

Constantemente no nosso dia a dia, tomamos decisões a partir de determinadas premissas. Podemos usar árvores para exemplificar tal processo e facilitar a tomada de decisão. Vejamos um simples exemplo a seguir de como representar esse processo.

Figura 8.27: Árvore de decisão.

Processo de busca

Vimos que as estruturas lineares permitem a realização de busca por elementos armazenados nelas. É um processo simples e útil. Entretanto, quando a quantidade de dados se torna demasiadamente grande (dezenas de milhares ou milhões), essas estruturas começam a ser um gargalo de processamento devido ao processo de visitar cada elemento da estrutura. Neste ponto, *árvores binárias ordenadas* são uma estrutura mais eficiente para realizar essa operação. Para exemplificar, vejamos a seguinte árvore binária ordenada:

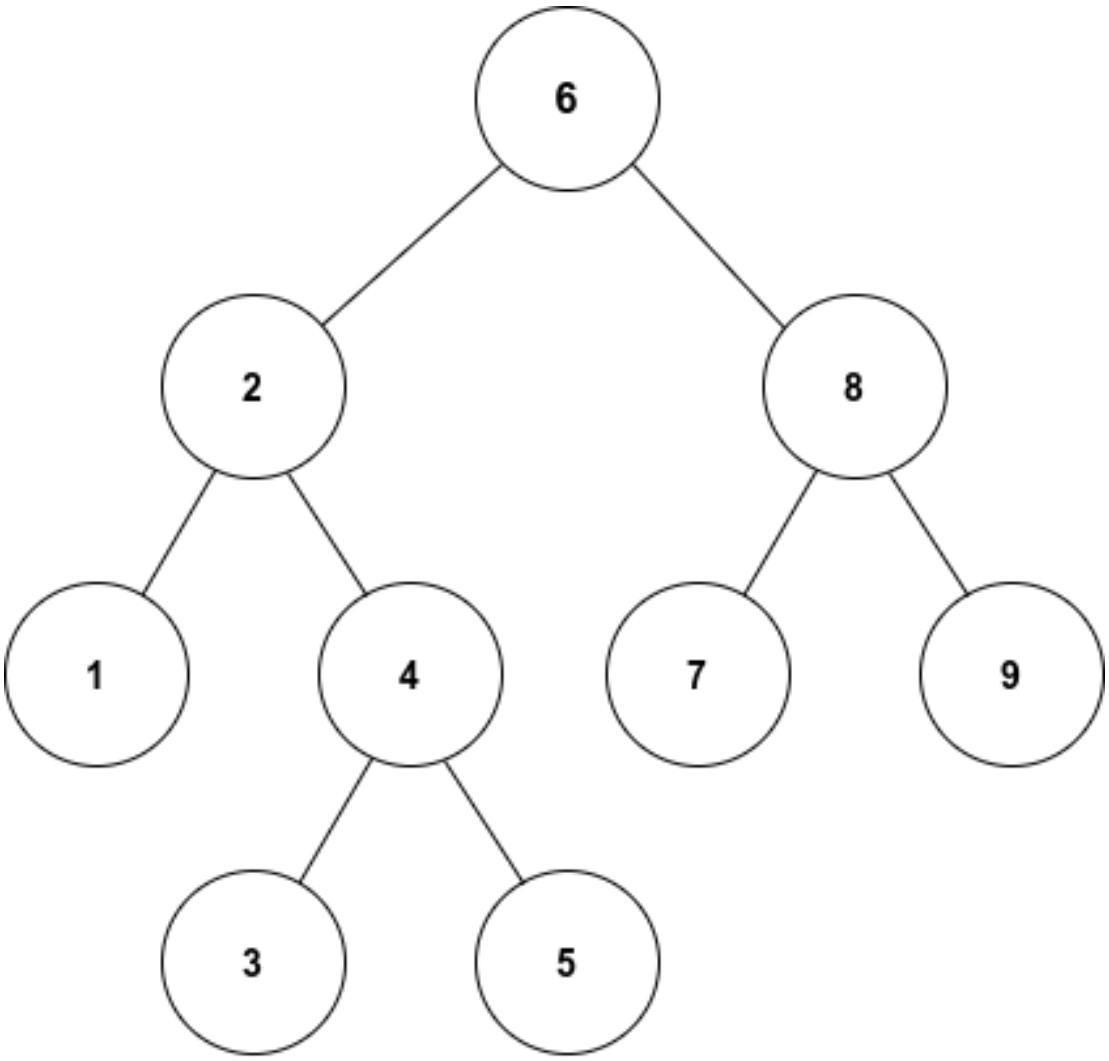


Figura 8.28: Árvore de busca.

Nessa árvore, para encontrar o elemento `7`, três comparações seriam necessárias — no caso, com `6`, `8` e `7`. Se esses dados estivessem em uma lista ordenada, precisaríamos de sete comparações. Ou seja, usando a ED árvore, diminuímos drasticamente a quantidade de verificações, o que torna o processo mais performático e mais rápido.

8.7 Exercícios

1) Faça alterações para os códigos de nossa *árvore clássica* usar um vetor, em vez de termos os três filhos fixos.

2) Faça alterações para o `update` de nossa *árvore clássica* não usar o `search`, mas, sim, ser recursivo.

3) Aproveitando os códigos de nossa *árvore clássica* (sem vetor), crie uma função que conta a quantidade de *nós* de nossa *árvore clássica*.

4) Aproveitando os códigos de nossa *árvore AVL*, crie uma função que verifica a profundidade de uma AVL.

5) Dadas as seguintes *árvores AVL*, faça o que se pede:

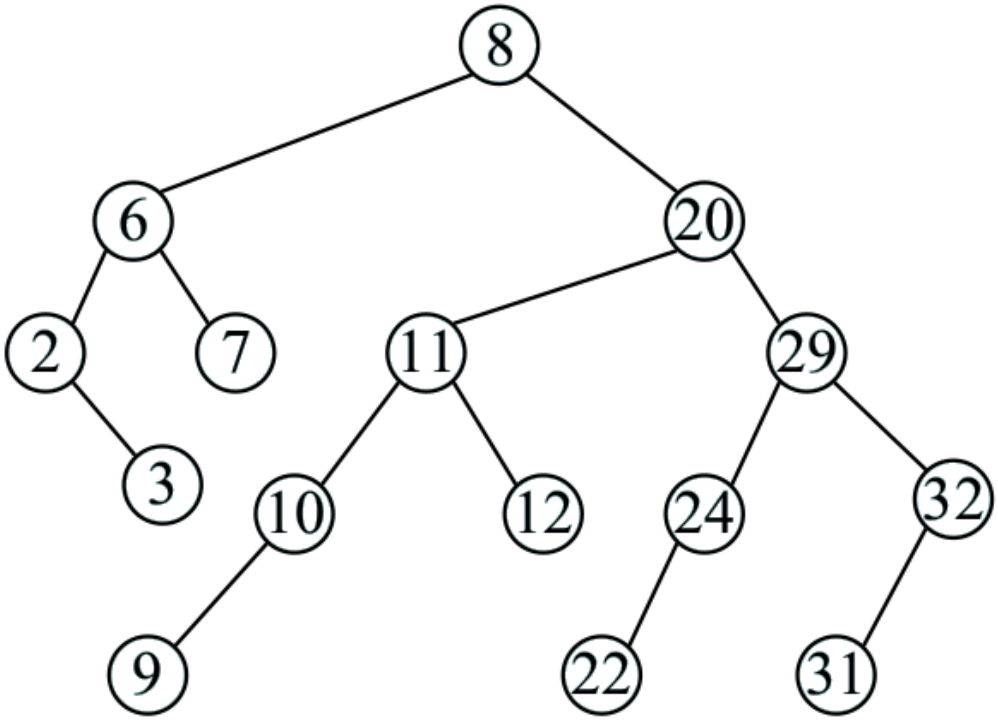


Figura 8.29: Árvore 1.

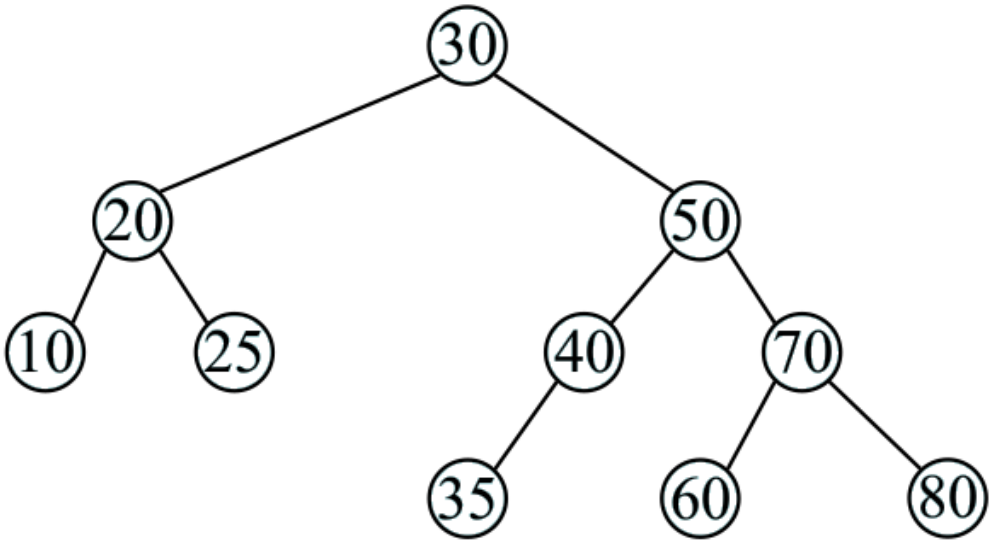


Figura 8.30: Árvore 2.

a) Remova os elementos `22`, `31`, `12`, `7` e `20` da Árvore 1.

b) Remova os elementos `40`, `25`, `50`, `10`, `35`, `30`, `20`, `70` e `60` da Árvore 2.

Observação: este não é um exercício de implementação, mas, sim, de fixação. Faça na mão com vários desenhos.

CAPÍTULO 9
Grafo

Neste capítulo, vamos abordar a última estrutura que faz parte das bases da disciplina de Estruturas de Dados: o **grafo**. Essa pode ser considerada a ED mais complexa que temos, mas este capítulo trará uma introdução suficiente para entendê-la e também fornecerá sólidos conhecimentos para estudos avançados.

9.1 Fundamentos

Podemos definir um *grafo* como:

> Uma estrutura de dados onde os elementos podem estar livremente relacionados entre si.

Essa definição deixa clara a natureza dos grafos: serem uma estrutura *relacional*. Os elementos são livres para se relacionar (se conectar) com quem for necessário. Não existe a limitação de *pai* e *filho* que as árvores possuem, por exemplo, onde sempre tínhamos uma direção (da *raiz* às *folhas*). Isso termina por agregar à complexidade citada no início do capítulo, pois essa liberdade de relacionamentos traz um aumento nas possibilidades de caminhos que podemos percorrer dentro dessa estrutura.

Devido a essa complexidade adicional, podemos achar que essa ED não é muito comum no nosso dia a dia, mas, sim, somente em avançadas necessidades computacionais, como redes neurais ou Inteligência Artificial (IA). Mas isso não é verdade. Vejamos a imagem a seguir, que é uma visão particular do Marco Zero de Recife.

Figura 9.1: Marco Zero de Recife.

Através da imagem anterior, podemos detectar que vários trajetos levam à Praça do Marco Zero. Se levarmos em consideração que cada esquina é um ponto de partida, podemos traçar vários trajetos. Para tornar isso mais claro, veja a imagem a seguir.

Figura 9.2: Marco Zero de Recife em grafo.

A imagem anterior demonstra que cada elemento (esquina) se conecta a quantos outros elementos (esquinas) forem necessários para criar um trajeto (rua) até a praça. A união desses elementos e trajetos formam um *grafo*. Ou seja, essa ED é muito mais comum do que podemos imaginar.

Para finalizar esta seção, posso dizer que exploramos esse exemplo para ilustrar o que é um grafo por ele ser similar ao *primeiro grafo*

de que se tem notícia na história da humanidade: o problema das sete pontes de Königsberg. Ele é considerado a primeira vez que um *grafo* foi detectado e sua solução deu origem à *Teoria dos Grafos*. Basicamente, esse problema desafiava uma pessoa a passar por todas as pontes sem repetir nenhuma delas.

Para saber mais sobre esse problema, acesse o primeiro dos links fornecidos a seguir. Caso tenha uma curiosidade maior, acesse o segundo link e veja a localização exata da ilha na cidade de Königsberg que originou esse problema.

- Sete pontes de Königsberg: https://pt.wikipedia.org/wiki/Sete_pontes_de_K%C3%B6nigsberg
- Localização: https://www.google.com.br/maps/@54.7070333,20.5105764,1150m/data=!3m1!1e3

Terminologias

Assim como na ED *árvore*, os *grafos* possuem um conjunto de termos inerentes a eles, que auxiliam seu entendimento e uso. Para facilitar as explicações, levemos em consideração a imagem a seguir, que possui três grafos em sua forma mais comum de representação gráfica.

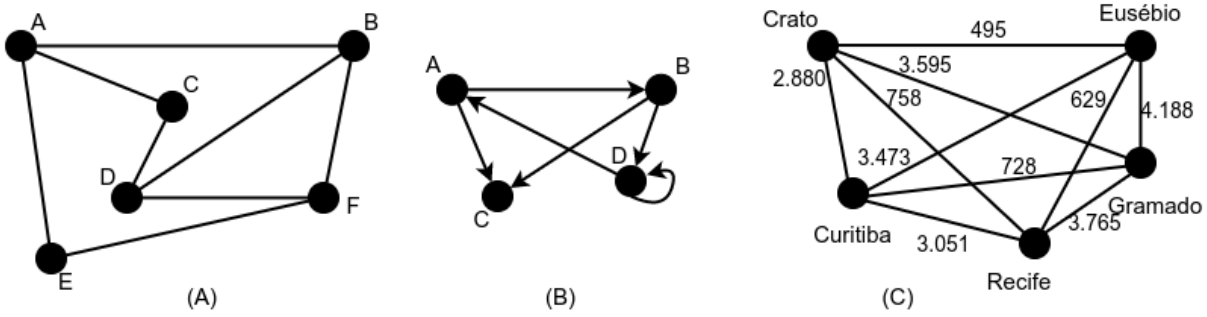


Figura 9.3: Grafos.

- **Vértice (nó)**: assim como as árvores, os grafos também possuem seu elemento base, os quais são chamados de **vértices**. Assim como os *nós* das árvores, eles podem armazenar dados e servem para montar a estrutura do grafo. Na figura anterior, as letras em maiúsculo (grafos (A) e (B)) e os nomes das cidades (grafo (C)) são *vértices*;

- **Aresta**: são as *conexões entre os vértices*. Essas conexões também estão presentes nas árvores, mas nelas são meras coadjuvantes; já nos grafos, podem ter valores e direções. Na figura anterior, as arestas conectam os vértices, que são os pontos de referências dos grafos. Elas podem ter algumas características a mais, que são exploradas a seguir;

    - **Peso**: são os *valores que as arestas podem possuir*. A presença ou ausência de pesos influencia no *tipo* do grafo, como veremos na seção *Tipos de grafos*. No grafo (C) da figura anterior, os pesos são as *distâncias* entre as cidades;
    - **Laço (loop)**: é quando o *vértice* de um grafo possui uma **aresta que leva a ele mesmo**. Isso ocorre no grafo (B) com o vértice D. É válido dizer que esse tipo de aresta só faz sentido existir em *grafos direcionados*. Na seção *Tipos de grafos* abordaremos esse tipo de aresta novamente;
- **Vértice adjacente (vizinho)**: é o *vértice* ou *vértices* alcançáveis a partir de um determinado vértice. No grafo (B), o vértice A tem como adjacentes os vértices B e C. Já nos grafos (A) e (C), são todos vértices alcançáveis a partir de um determinado vértice;

- **Grau (valência) de um vértice**: é a quantidade de *arestas* que um *vértice* possui. Esse grau pode ser de *de fora* ou *de dentro*. No primeiro, são as arestas que *saem* do vértice e no segundo, as arestas que *chegam* ao vértice. Esse tipo de diferenciação é válido para grafos direcionados. Para grafos não direcionados, resume-se às arestas conectadas ao vértice. Na seção *Tipos de grafos* veremos a diferença entre esses grafos;

- **Caminho (trajeto)**: são as *arestas* percorridas para chegar de um determinado vértice a outro. É nesse(s) caminho(s) que a Teoria dos Grafos foca, pois é aí onde estão seus maiores desafios.

9.2 Operações

Assim como as árvores, os grafos também possuem operações e formas de navegação. Estas são um pouco diferentes em relação às das árvores, mas têm a mesma finalidade: modificar e navegar pela ED. São quatro operações e duas formas de navegação. A lista a seguir explica as operações e, na sequência, veremos as técnicas.

- `ADD` : adiciona um novo vértice ao grafo;
- `JOIN` : após o vértice ser adicionado, é necessário conectá-lo a outro vértice, criando uma *aresta*. Essa é a finalidade dessa operação;
- `REMOVE` : exclui um vértice do grafo, acompanhado de suas arestas;
- `ADJACENT` : identifica os vértices que possuem arestas com um determinado vértice.

A depender do tipo de grafo, direcionado ou não e com pesos ou não, as operações podem sofrer modificações, mas sem perderem a sua essência.

Busca em profundidade

Para essa forma de navegação, inicialmente escolhe-se um vértice de forma aleatória. Após isso, inicia-se o processo de navegação passando por todos os vértices do grafo — por isso o nome *profundidade*, pois esse processo vai até o vértice "mais profundo" do grafo. Cada vértice visitado é marcado como *processado*, para evitar repetições e para que a busca não entre em loop. Essa

navegação é mais simples de implementar, porém menos eficiente, já que essa busca ao vértice mais profundo descarta muitos outros vértices pelo caminho. Quando se chega a esse vértice mais profundo, retorna-se ao ponto de partida para processar os anteriormente descartados.

Busca em extensão ou largura

Para essa outra forma de navegação, também escolhe-se aleatoriamente um vértice de partida. Entretanto, ele não vai passando de forma aleatória por todos os demais vértices após isso. Ela sempre passa primeiro por todos os *vértices adjacentes* ao vértice corrente. Depois, movimenta-se para os adjacentes dos adjacentes, e assim por diante. Por isso o nome "em largura", pois essa busca vai expandido a cada vértice visitado. Assim como a navegação em profundidade, vértices visitados são marcados como *processados* para evitar loops. Essa navegação é mais difícil de implementar, mas tem uma performance um pouco melhor que a *busca em profundidade*, devido ao fato de evitar retrocessos.

Ambas as navegações são consideradas de "força bruta", por navegarem de forma aleatória pelos vértices. Todavia, a *busca em largura* é ligeiramente melhor quando se tem conhecimento prévio de que os vértices a serem encontrados estão relativamente próximos ao vértice de partida.

9.3 Tipos de grafos

Veremos aqui os tipos mais relevantes de grafo, que são essenciais para o entendimento dessa ED.

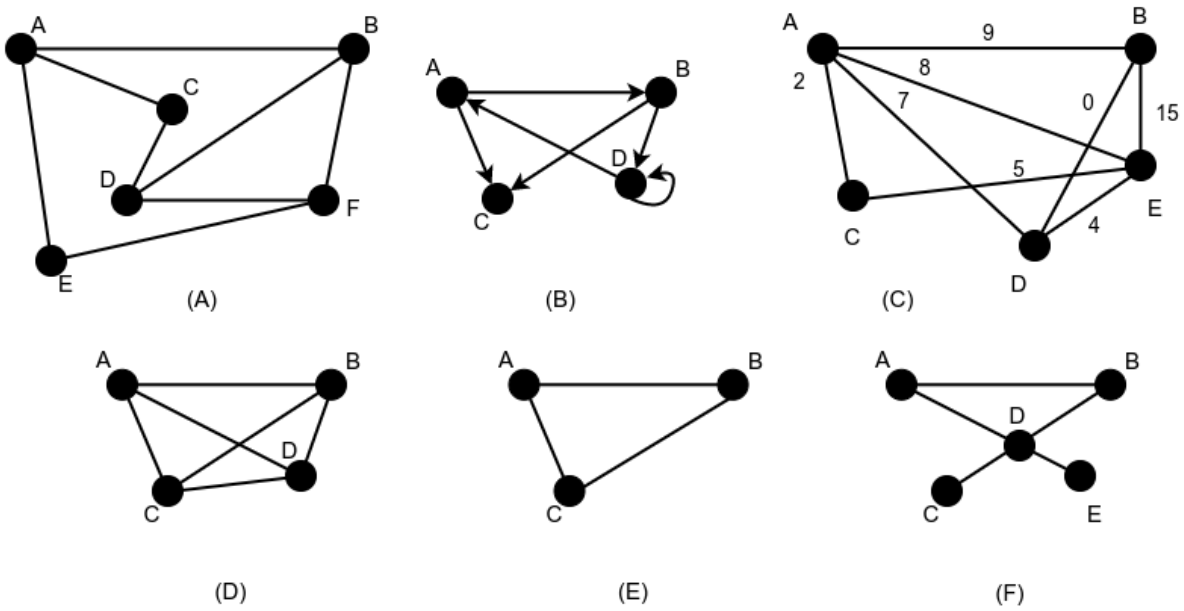


Figura 9.4: Tipos de grafos.

- **Simples**: é um grafo não direcionado, que não possui *laços* e que possui somente uma *aresta* entre quaisquer dois *vértices*, ou seja, não tem *arestas paralelas*. Em exceção ao grafo `(B)`, todos os demais são exemplos desse tipo;
- **Completo**: é um grafo em que cada *vértice* está conectado a todos os demais *vértices*. Vemos este nos grafos `(D)` e `(E)`;
- **Regular**: é um grafo no qual todos os *vértices* têm o mesmo *grau*. Vemos este novamente nos grafos `(D)` e `(E)`. Assim, note que todo *grafo regular* é, também, um *grafo completo*, e o inverso também é válido;
- **Não direcionado**: é um grafo no qual as *arestas* não possuem um sentido. Em exceção do grafo `(B)`, todos os demais são desse tipo. Neste caso, é indiferente avaliar se o sentido é `(A,B)` ou `(B,A)`;
- **Direcionado (dígrafo)**: é um grafo em que as *arestas* possuem um sentido que influencia nos trajetos que podem ser realizados. Isso ocorre no grafo `(B)`. Neste caso, é diferente

avaliar `(A,B)` ou `(B,A)`: `(B,A)` não é um trajeto permitido pelo grafo `(B)`, ao contrário de `(A,B)`;

- **Grafo misto**: é um grafo no qual suas *arestas* podem ter um sentido ou não. No caso, é uma mistura do grafo `(A)` e do grafo `(B)`;
- **Conexo**: é um grafo que possibilita um caminho de qualquer um de seus *vértices* para qualquer outro *vértice* do grafo. Assim, somente o grafo `(B)` não o é, pois do *vértice* `C` não conseguimos ir para nenhum outro *vértice*;
- **Ponderado**: é um grafo no qual as *arestas* possuem *pesos*. Vemos isso no grafo `(C)`;
- **Cíclico**: é um grafo no qual, a partir de qualquer *vértice* de partida, consegue-se passar por todos os demais, sem repeti-los, e atingir o *vértice* de partida. Vemos esse tipo nos grafos `(C)`, `(D)` e `(E)`;
- **Acíclico**: é um grafo no qual não se é possível ter um ciclo. Vemos esse no grafo `(F)`;
- **Árvore**: é isso mesmo, uma árvore é um tipo de grafo. Porém, é um grafo *simples*, *acíclico* e *conexo*. Vimos esse tipo em todos os exemplos do capítulo de *árvores*.

9.4 Representações

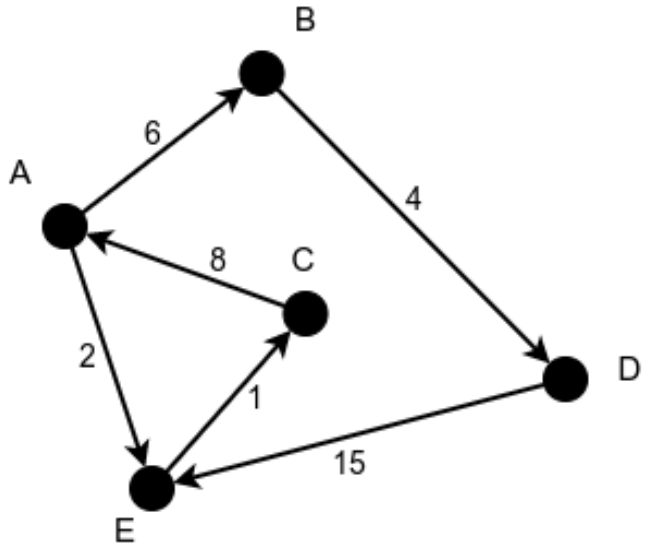
As imagens que vimos até agora são as representações visuais de grafos, mas quando vamos codificar um *grafo* em alguma linguagem de programação, temos três opções de representá-lo, as quais serão apresentadas a seguir.

Matriz de incidência

Essa representação é mais alinhada com *grafos orientados ponderados*, mas isso não significa que não possa ser usada com grafos *não orientados* e *não ponderados*. Nessa matriz, as colunas são as *arestas* do grafo, mais especificamente seus *pesos*, e as linhas são seus *vértices*. Por fim, também devemos definir como representar a *incidência* das arestas aos vértices, que geralmente usa o seguinte padrão:

- `-1` para representar uma origem de aresta ou um laço;
- `0` para representar vértices que não possuem uma aresta de origem ou destino;
- `1` para representar um destino de aresta.

A imagem a seguir demonstra essa representação.



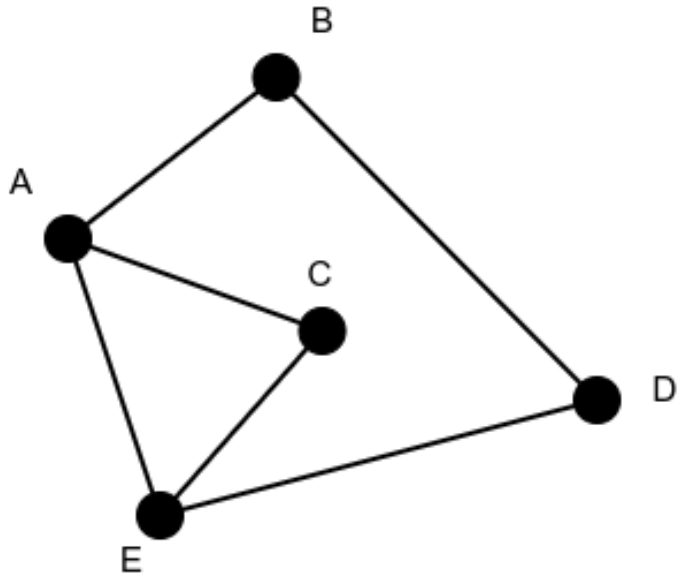
	6	4	15	1	8	2
A	-1	0	0	0	0	-1
B	1	-1	0	0	0	0
C	0	0	0	1	-1	0
D	0	1	-1	0	0	0
E	0	0	1	1	0	1

Figura 9.5: Matrizes de incidência.

Matrizes de adjacência

Um grafo representado por essa abordagem utiliza uma matriz onde os *vértices* são dispostos em linhas e colunas. Então, nas relações em que existir uma aresta (ou seja, vértices adjacentes), é feita uma marcação, geralmente com o número `1`; onde não existir, marca-se com o número `0`. Assim, quando vemos `M[A,B]`, significa que existe

uma aresta ligando os vértices A e B. A imagem a seguir demonstra essa representação.



	A	B	C	D	E
A	0	1	1	0	1
B	1	0	0	1	0
C	1	0	0	0	1
D	0	1	0	0	1
E	1	0	1	1	0

Figura 9.6: Matrizes de adjacência.

Listas de adjacência

Essa representação pode utilizar as EDs *vetor* e *lista*, a depender de uma maior necessidade de inclusão de vértices no grafo. Essas estruturas podem ser ligadas entre si da seguinte forma: cada vértice do grafo é armazenado em um *vetor*. Assim, cada posição desse vetor aponta para uma *lista* que contém os vértices adjacentes ao vértice armazenado. A imagem a seguir expõe essa representação.

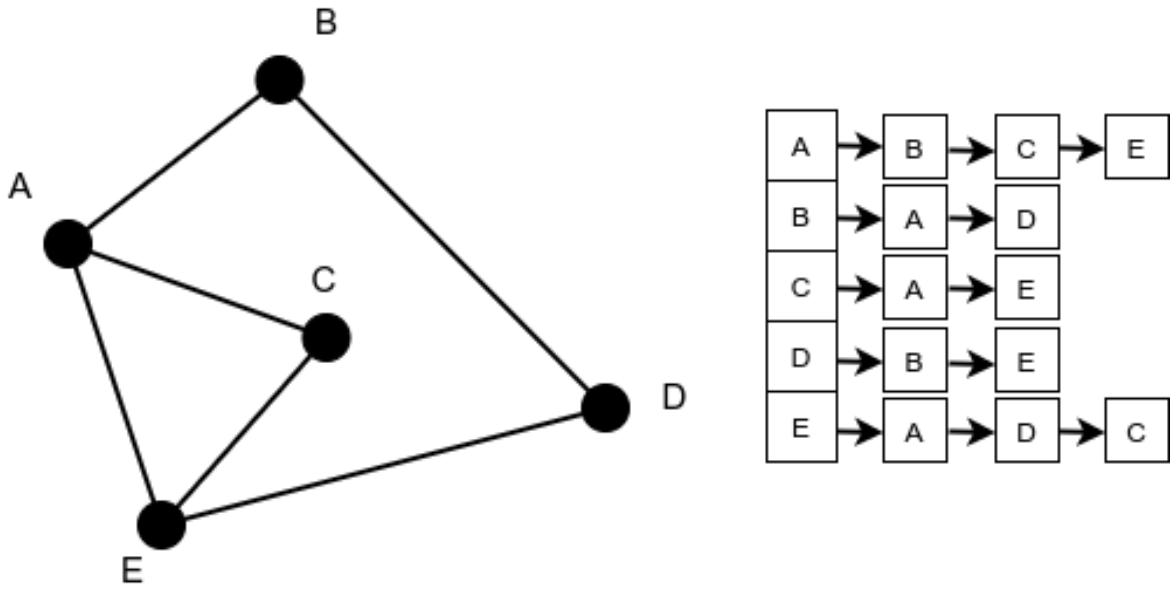


Figura 9.7: Listas de adjacência.

9.5 Problemas clássicos

Por ser um assunto de grande relevância e aplicação a várias áreas, existem alguns problemas clássicos sobre grafos que auxiliam a entendê-los e difundi-los. A seguir, apresentaremos outros três famosos problemas.

Problema do caminho mínimo

Esse problema tem por finalidade minimizar o custo de travessia de um grafo entre dois vértices: a origem e o destino. Para isso, o cálculo desse custo leva em consideração a soma dos *pesos* de cada *aresta* (e consequentemente *vértices*) a ser percorrida entre o vértice de origem e o de destino. O Google Maps utiliza a solução desse problema para fornecer sempre a menor rota.

Caixeiro viajante

Esse problema tem por finalidade determinar qual a menor rota a ser percorrida através de um determinado conjunto de cidades, sendo que se deve visitar cada cidade apenas uma vez e depois retornar à cidade de origem. Uma aplicabilidade da solução para esse desafio é na área de logística, para otimizar rotas de entregas de produtos ao percorrer o menor caminho possível, com redução do tempo necessário para transporte e de possíveis custos de combustível.

Teorema das quatro cores

Esse problema desafia quem deseja solucioná-lo a somente usar quatro cores para colorir as regiões ou estados de qualquer mapa de um país, de modo que duas regiões ou estados adjacentes (vizinhos) não tenham a mesma cor.

9.6 Algoritmos importantes

Muitos são os algoritmos que existem para solucionar os problemas citados na seção anterior ou mesmo outros problemas. A seguir, vejamos os dois mais famosos.

Dijkstra

Edsger Dijkstra é um cientista da computação holandês que entre 1956 e 1959 criou o algoritmo que soluciona o problema do caminho mais curto em um *grafo orientado* ou *não orientado*, estes sendo *ponderados* e com *pesos* não negativos. Esse algoritmo inicia o processo em um vértice inicial e, a partir deste, navega pelas arestas de menor peso para os vértices adjacentes. Esse processo se repete para os adjacentes dos adjacentes até se chegar ao destino. Arestas

que foram avaliadas em processamentos anteriores são descartadas durante o processo de verificação.

Vizinho mais próximo

Esse algoritmo é um dos passos para se resolver o problema do caixeiro viajante. Ele começa o processo a partir de um vértice inicial, escolhido aleatoriamente. Após isso, identifica-se a *aresta de saída* de menor *peso* do vértice atual que leve a um vértice não visitado. Assim, marca-se o vértice atual como "visitado" e torna o vértice de destino o atual. Se todos os vértices do grafo tiverem sido visitados, encerra-se o processo; se não, devemos novamente identificar a aresta de saída de menor peso do vértice atual e, assim, repetir o processo até visitarmos todos os vértices.

A sequência dos vértices marcados como "visitados" é o resultado do algoritmo. Esse algoritmo é simples de implementar, mas não é 100% eficiente por algumas vezes não obter o melhor caminho (o menor) como resultado. Isso vem do fato de ser considerado um "processamento guloso", ou seja, às vezes já considera um resultado como bom (menor *peso*), sem ter verificado todas as outras possibilidades.

9.7 Exemplos de uso

Não diferente das estruturas de dados vistas anteriormente, o grafo possui uma grande quantidade de aplicabilidades. Vejamos algumas.

Redes sociais

Tenho certeza de que, assim como eu, você possui alguma rede social. Lá, temos nossas conexões, que são amigos, conhecidos, *influencers*, entre outros. Estas conexões também possuem suas próprias conexões, que podem ter algumas conexões em comum

entre si e também conosco. Para ilustrar isso, vejamos a imagem a seguir, que é uma visão parcial de minhas conexões no LinkedIn.

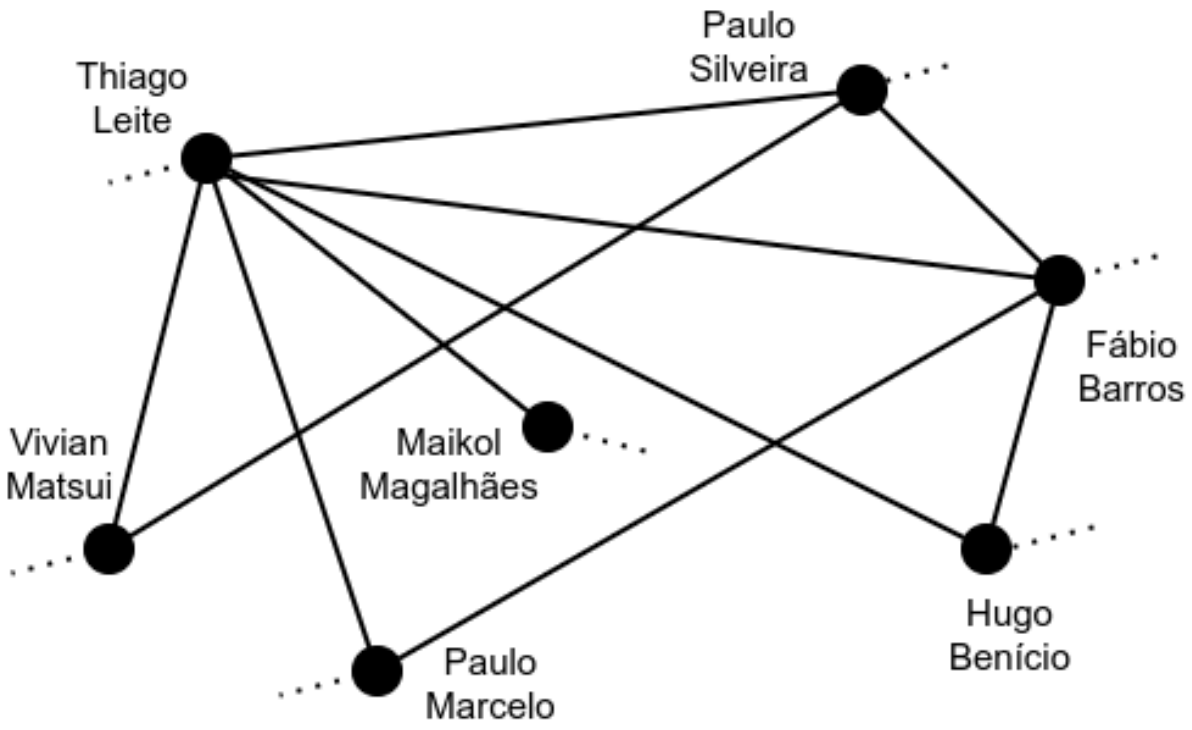


Figura 9.8: Conexões LinkedIn.

Com essa imagem, vemos claramente que as redes sociais são uma aplicação de grafos. Nesta, cada pessoa será um *vértice* do grafo e as conexões entre as pessoas são as *arestas*.

Máquina de estados

Máquina de estados é um conceito ligado à matemática que tem como finalidade representar a mudança de estados de entidades quando elas sofrem um estímulo. Várias são suas aplicabilidades:

- Circuitos lógicos de computadores;
- Circuitos elétricos da física;
- Diagramas/Fluxos;

- Autômatos finitos.

Para deixar mais claro, vejamos o seguinte fluxo da realização de uma compra:

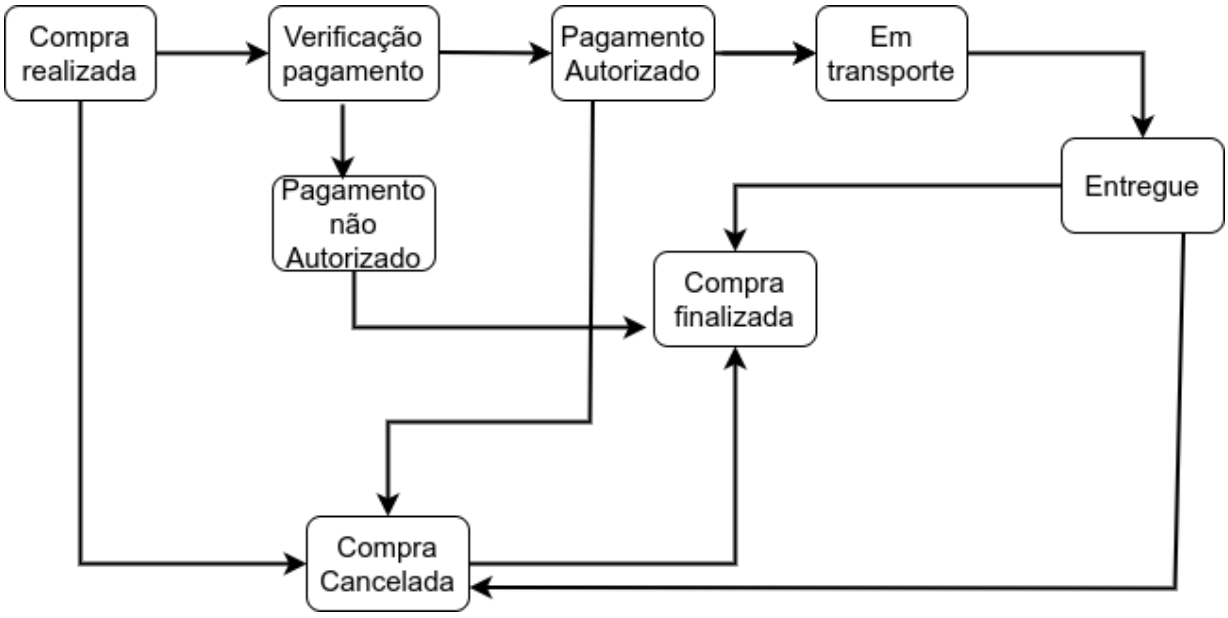


Figura 9.9: Fluxo de uma compra.

Nesse fluxo, vemos que temos operações (vértices) que são conectadas por setas (arestas) para outras operações. Essas operações têm uma sequência predefinida de execução. Podemos notar que esse fluxo é um *grafo direcionado*, onde temos um *vértice de início* fixo (`Compra realizada`) e um *vértice de destino* fixo (`Compra finalizada`).

Geralmente, *máquinas de estados* são grafos simples, pois, além de serem direcionados, possuem poucos caminhos possíveis a serem percorridos.

Google Maps

Quem nunca usou essa aplicação certamente não habita nosso planeta. O Google Maps é um ótimo exemplo do uso de grafos. Toda vez que traçamos uma rota, ele cria um ou mais caminhos entre a origem e o destino (os quais são dois *vértices*). Entre eles, temos

várias esquinas, que serão *vértices* intermediários. Temos também ruas que ligam essas esquinas, que seriam as *arestas*. Ou seja, um belo grafo!

O Maps é um exemplo tão bom que temos ainda a seguinte situação: se escolhermos uma rota *a pé*, veremos que o Maps vai sugerir só uma opção, que será a menor (melhor caminho). Entretanto, se colocarmos a rota *de carro*, veremos que teremos mais de uma opção, com distâncias diferentes. Isso ocorre porque, de carro, teremos que seguir os sentidos das ruas, ou seja, teremos um *grafo direcionado*. No caso da opção *a pé*, temos um *grafo não direcionado*, pois podemos ir da forma que quisermos, sem nos preocupar com *mão* ou *contramão*.

9.8 Só isso sobre grafos?

Na verdade, não. Devido aos grafos serem uma generalização das *árvores*, muito mais pode ser explorado sobre esse assunto. Mas se o nosso capítulo sobre *árvores* já foi tão extenso, imagine se explorássemos a fundo os grafos. Seria um outro livro dentro deste livro! Mas é justamente isso: um livro. A própria Casa do Código possui um livro sobre grafos: *Teoria dos Grafos: Uma abordagem prática em Java* (ISBN: 978-65-86110-51-7), escrito pelo nosso colega João Paulo e que possui um conteúdo mais aprofundado do que este capítulo. Caso precise de algo mais avançado, aconselho a leitura desse livro. Além dele, deixo outras recomendações de livros na seção *Referências bibliográficas*.

Capítulo 10
Outras estruturas de grande relevância

Embora as estruturas apresentadas do Capítulo 3 ao Capítulo 9 sejam as mais conhecidas e amplamente utilizadas, ainda existem três outras que não fazem parte da Teoria das Estruturas de Dados, mas que também são bem conhecidas e não é difícil as vermos em uso em diversos softwares. Devido a isso, vamos explorá-las neste capítulo.

10.1 Set (conjunto)

Essa estrutura de dados tem como principal característica não permitir a repetição de elementos. A partir disso, podemos ver que ela é "materialização" do conceito matemático de *conjuntos*. Devido a isso, a ED *set* também tem um importante comportamento dos *conjuntos*: seus elementos não possuem uma ordem, ou seja, são armazenados de forma aleatória. Essas são as mais relevantes características sobre a ED *set*, e isso termina por disponibilizar uma ED de simples entendimento e de uso intuitivo. Na Parte 2 deste livro, exploraremos essa ED nas linguagens abordadas.

10.2 Tabela de dispersão (Hashtable)

Muitas vezes precisamos ter acesso a elementos em uma estrutura de dados de forma rápida. Sabemos que, ao usar *vetores*, conseguimos esse acesso rápido, mas teremos a desvantagem de ter o tamanho fixo da estrutura. Então podemos pensar em usar uma *lista*, que tem tamanho variável. Todavia, teremos um processo lento

de leitura, pois teremos que navegar por ela — de elemento em elemento — para encontrar o elemento desejado. Baseado nessas informações, talvez se cogite usar uma *árvore*, que tem o dinamismo desejado e o acesso rápido, caso se use uma *árvore binária ordenada*, por exemplo. Mas ela pode se tornar lenta, caso sua profundidade seja muito grande, além de consumir mais memória para percorrê-la, devido à recursividade.

Ou seja, existem casos em que o acesso rápido é primordial e as estruturas clássicas que conhecemos talvez não sejam a melhor opção, devido ao tempo de pesquisa ou à memória consumida. Nesses casos, podemos usar a **Tabela de dispersão (*Hashtable*)**. Ela é uma ED auxiliar que possui um acesso posicional muito rápido e ainda com a vantagem de ter tamanho dinâmico. Para isso, essa ED calcula o endereço (chave) de onde o elemento será armazenado a partir de uma função *hash*, que pode usar os dados do próprio elemento ou ser fornecida de forma livre. Dessa forma, sabe-se exatamente onde o elemento se encontra, o que permite um acesso mais rápido. Para ilustrar isso, levemos em consideração a imagem a seguir:

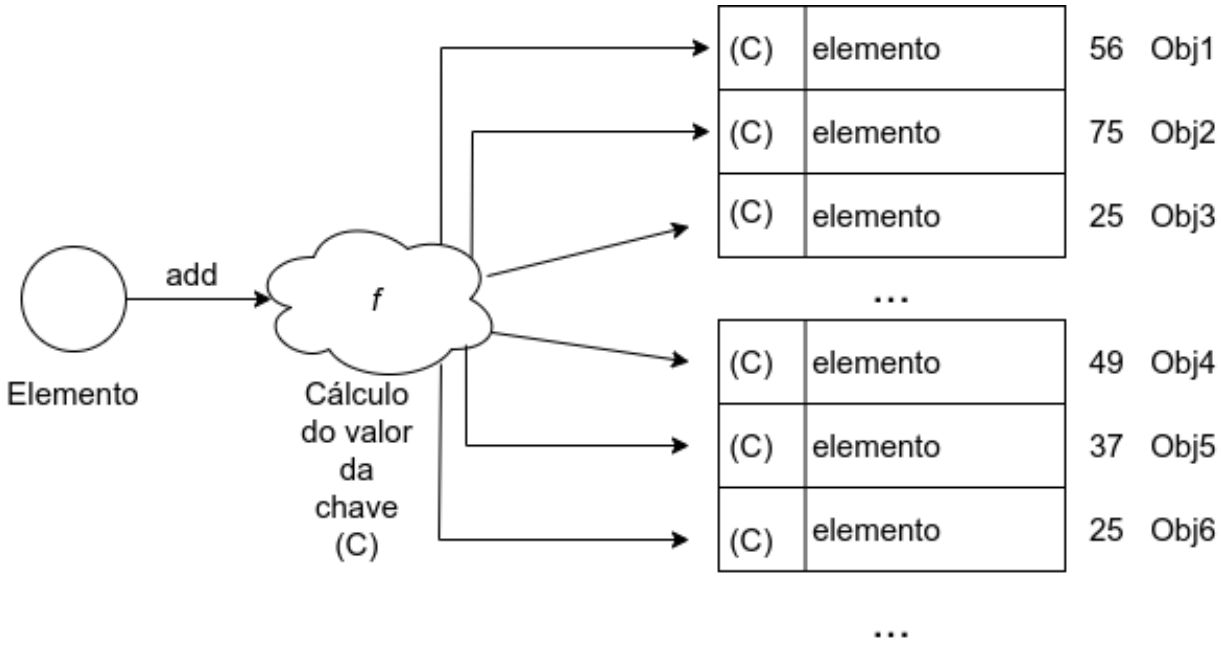


Figura 10.1: Funcionamento de uma Tabela de dispersão.

Na imagem anterior, vemos que, no momento em que cada elemento foi adicionado na tabela, seu *hash* foi calculado e armazenado junto a ele. Assim, quando se desejar acessar um desses elementos, usa-se sua chave, que lhe levará exatamente ao endereço específico, evitando passar por todos os outros elementos da tabela.

Uma observação relevante sobre a Tabela de dispersão é que nada impede de os valores *hash* coincidirem (no caso, os *hashs* 25 em nossa imagem). Isso é um problema, pois poderíamos ter dois elementos com a mesma chave. Mas existem algoritmos que auxiliam na geração de chaves únicas ou no tratamento de chaves repetidas, o que possibilita o uso dessa ED de forma ampla.

Para finalizar, muitas linguagens usam essa ED sob o nome de **mapas** ou **dicionários**. Novamente, na Parte 2 deste livro, exploraremos essa ED nas linguagens abordadas aqui.

10.3 Heap

Também chamada de *Binary Heap*, essa ED é uma variação de uso de *árvores binárias*. Para uma *árvore binária* poder ser considerada um *heap*, uma regra deve ser atendida: valores em *nós pais* devem ser sempre maiores ou iguais aos *nós filhos* (*max-heap*) ou sempre menores ou iguais aos *nós filhos* (*min-heap*). Essa é a chamada **propriedade *heap***.

Devido a essa característica, essa ED é muito utilizada como uma implementação de *filas de prioridade*. Ou seja, elementos de maior ou menor prioridade vão sendo removidos sempre a partir da raiz da árvore. Após essas remoções, é necessário reorganizar a árvore, realizando rotações, para manter a propriedade *heap* válida.

Para exemplificar isso, vejamos as duas *árvores* a seguir:

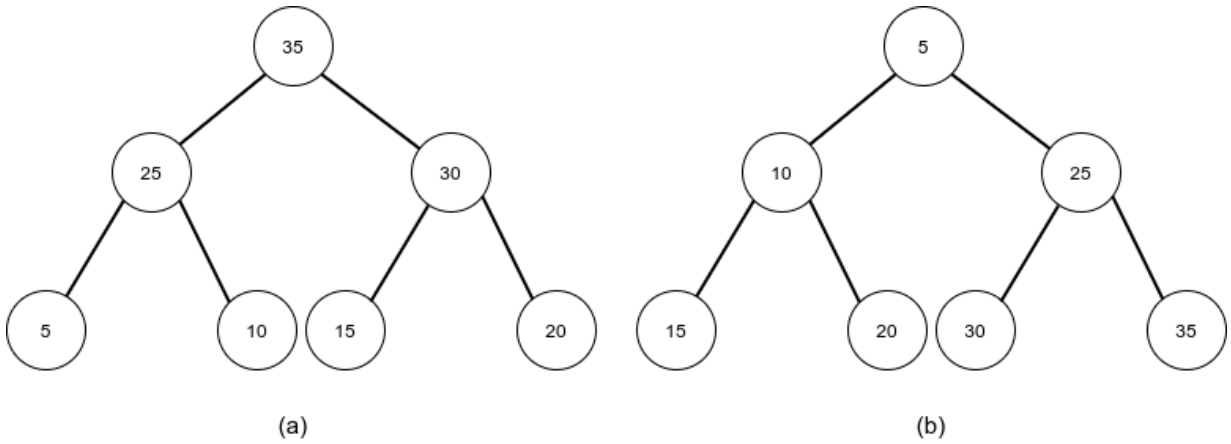


Figura 10.2: Heaps.

Na imagem anterior, em `(a)` temos uma *max-heap*, pois podemos notar que os *nós pais* (no caso, `35`, `25` e `30`) são sempre maiores que seus *nós filhos*. Já em `(b)`, temos uma *min-heap*, pois `5`, `10` e `25` são sempre menores que seus *nós filhos*.

Parte 2: O mundo real

Após as entranhas das estruturas de dados terem sido exploradas, chega a hora de usá-las de fato. Entretanto, no "mundo real", onde reinam linguagens Orientadas a Objetos (OO) ou Funcionais, usar ponteiros realmente não é mais a única opção. Assim, nesta parte do livro, as estruturas de dados clássicas serão apresentadas nas principais linguagens do mercado: Java, C#, Python e JavaScript. Essas linguagens abstraem o uso de ponteiros, o que nos ajuda a focar no problema a resolver, e não em criar a estrutura de dados em si. Além disso, algumas outras estruturas de dados úteis que estas linguagens disponibilizam também serão apresentadas. Então, se você desenvolve especificamente em Java, C#, Python ou JavaScript, aconselho focar nos respectivos capítulos.

Aqui veremos as facilidades que tais linguagens fornecem para manipular as estruturas e como utilizá-las. Independente de desenvolver back-end ou front-end, conhecer bem as estruturas de dados é muito importante para tirar mais proveito de cada linguagem. Algumas das estruturas estão presentes de forma nativa em suas linguagens, outras precisam de componentes de terceiros e outras são implementadas na linguagem específica. Mas em todos os casos, elas serão explicadas. No fim, veremos que manipular as estruturas de dados apresentadas neste livro através de linguagens OO é bem mais prático.

Capítulo 11
Estruturas de Dados em Java

11.1 Vetor e Matriz

Embora Java disponibilize a classe *Vector*, essa não é de fato um *vetor*. Ela se assemelha mais a uma *lista*. Mas a verdade é que essa classe não é usada na prática, e a própria documentação de Java

aconselha a usar outras classes em vez desta. Assim, vamos explorar *vetores* e *matrizes* de formar nativa.

> O código completo está disponível no link: https://github.com/thiagoleiteecarvalho/ed-Java.git. Clone-o para um melhor estudo.

- Para criar um *vetor* e uma *matriz*, fazemos:

```
int[] v = new int[5];
int[][] m = new int[5][5];
```

No código anterior, criamos um *vetor* chamado `v` e uma *matriz* chamada `m`. Ambos são estruturas que armazenam inteiros (`int`). Em Java, para definirmos estas estruturas, usamos a forma a seguir:

- `<tipo>[] nome <?=valorInicial?>;` ou `<tipo> nome[] <?=valorInicial?>;`
- `<tipo>[][] nome <?=valorInicial?>;` ou `<tipo> nome[][] <?=valorInicial?>;`

Das duas definições anteriores, a primeira opção é a mais usual. Outras opções de criar e já inicializar *vetores* e *matrizes* são:

```
int[] v1 = {10,25};
int[] v2 = new int[] {1,2,3};
int[][] m1 = new int[][] {{1,2},{3,4}};
int[][] m2 = {{5,6},{7,8}};
int[][][] m3 = new int[][][] {{{1,2},{3,4}}, {{5,6},{7,8}}};
int[][][] m4 = {{{9,10},{11,12}}, {{13,14},{15,16}}};
```

Vale ressaltar que, nas opções anteriores, quando os valores iniciais já são definidos na criação do vetor e da matriz, não é necessário informar os tamanhos, como foram feitos nas primeiras definições.

- Para *colocar elementos* em nosso vetor ou matriz, fazemos:

```
v[0] = 70;
v[3] = 45;
m[0][0]= 65;
m[3][2]= 17;
```

Nos códigos anteriores, vemos que precisamos fornecer a posição dentro da estrutura cujo valor desejamos armazenar. Lembremos que Java é *zero-based*.

- Para *acessar elementos* em nosso vetor ou matriz, fazemos:

```
int iV = v[0];
int iM = m[0][0];
```

No código anterior, vemos que a forma de acessar os valores é bem parecida com a de armazená-los. Assim, `iV` e `iM` recebem os valores `70` e `65`, respectivamente.

- Para saber a *quantidade de elementos* de nossos vetores e matrizes, fazemos:

```
v.length;
m.length;
```

O vetor `v` foi criado com cinco posições, então `v.length;` tem como resultado `5`. Já a matriz `m` foi definida com `[5][5]`. Nesse caso, o resultado de `m.length;` deveria ser `25`, mas não é. Isso ocorre porque, como vimos no Capítulo 3, *Vetor e Matriz*, uma matriz é um "vetor de vetores". Então, quando executamos `m.length;`, é similar a `m[0].length;`, que retorna `5`. Ou seja, é o tamanho do *vetor* dentro da posição `0` de `m` (`m[0]`).

Para sabermos de fato o tamanho de uma matriz, todos os índices existentes na definição de sua primeira coluna devem ser somados. Para nossa matriz `m`, o tamanho seria: `m[0].length + m[1].length + m[2].length + m[3].length + m[4].length`. Essa soma resultaria em `25`.

Para finalizar, em Java, nos *vetores* e *matrizes*, não existe a ideia de excluir elementos. Isso vem do fato de estas EDs sempre serem uma

estrutura de tamanho fixo. O máximo que se faz é atualizar a posição para um valor que não tenha significado para o problema em resolução.

11.2 Pilha

A classe `Stack` é a implementação da ED *pilha* em Java. Nada mais óbvio, pois "pilha" em inglês é "*stack*". Vejamos a seguir como usar essa classe.

- Para *criar* uma pilha:

```
Stack<String> p = new Stack<>();
```

No código, criamos uma pilha chamada `p`, que armazena caracteres (*string*), como "A", "B", "CD" etc.

> Vale ressaltar que usamos *generics* para especificar o tipo de dado da nossa pilha. Assim, o `<>` é responsável por "generalizá-la". Para aprender mais sobre *generics*, acesse: https://docs.oracle.com/javase/tutorial/extra/generics/index.html .

- Para *empilhar elementos* (`push`) em nossa pilha, fazemos:

```
p.push("F");
p.push("L");
p.push("A");
```

- Para saber qual elemento está no *topo* (`top`) de nossa pilha, fazemos:

```
p.peek();
```

Como resultado dessa execução, obteremos o valor `A`.

- Para *desempilhar* (`pop`) um elemento de nossa pilha, fazemos:

```
p.pop();
```

A execução desse código nos retorna o valor `A`. Assim, a pilha possuirá agora somente os valores `F` e `L`.

- Para saber a *quantidade de elementos* de nossa pilha, fazemos:

```
p.size();
```

Inicialmente, após os três *pushes*, o tamanho era `3`, mas como executamos um *pop*, o tamanho atual é `2`.

- Para saber se a pilha está *vazia*, fazemos:

```
p.empty();
```

Como sabemos que seu tamanho atual é `2`, o resultado dessa execução é `false`.

- Adicionalmente, essa classe em Java fornece uma operação chamada `search`, que procura um elemento dentro da pilha. Para usá-la, fazemos:

```
p.search("F");
```

O resultado dessa execução muda de acordo com os elementos empilhados. Como dois elementos restaram empilhados na sequência, `F` e `L`, o resultado é `2`. Se um novo elemento for empilhado, o resultado será `3`. Vemos que `search` considera a distância do elemento desejado ao *topo* da pilha – a distância não é *zero-based*. Se o elemento não existir na pilha, `-1` é retornado.

Embora tenhamos visto a classe `Stack`, Java aconselha usar uma classe que implementa a interface `Deque`, como a `ArrayDeque` ou `LinkedList`, pois ambas proveem também o comportamento LIFO.

Ficaria `Deque<Integer> stack = new ArrayDeque<Integer>();`, por exemplo. Mas saber que existem as duas possibilidades ajuda a manter códigos mais antigos, pois a classe `Stack` está presente no Java desde sua primeira versão, enquanto a interface `Deque` e suas implementações estão disponíveis somente a partir de Java 6.

Tenhamos em mente que usar `Stack` não é errado, pois nem mesmo depreciada esta classe é. É apenas um conselho. Tanto `ArrayDeque` como `LinkedList` serão apresentadas no decorrer deste capítulo, então deixemos suas explicações para adiante. Apenas mostraremos aqui um de/para das operações da pilha para os métodos de `Deque` implementados nestas classes.

Método pilha	Equivalente método Deque
push(e)	addFirst(e)
pop()	removeFirst()
peek()	getFirst()

Para finalizar, embora existam esses de/para das operações, para se manter a abstração-alvo, no caso a ED Pilha, é aconselhável usar os métodos `push`, `pop` e `peek`, pois estão também estão disponíveis na interface `Deque` e suas respectivas implementações `ArrayDeque` e `LinkedList`.

11.3 Fila

A classe `LinkedList` é a implementação da ED *fila* em Java. Ela é uma classe de *General-purpose* (propósito geral) e provê o comportamento de mais de uma estrutura de dados. Aqui a exploraremos como uma *fila*.

- Para criar uma *fila*:

```
LinkedList<Integer> f = new LinkedList<>();
```

O código anterior cria uma *Queue (fila)* chamada `f`, que armazena inteiros (*Integer*), como `1`, `2` etc. Novamente usamos *generics*.

> `Integer` é uma *Wrapper Class*, uma classe que represente o tipo de dados primitivo `int`. Além desta, existem outras *Wrapper Class*. Para aprender mais sobre elas, acesse https://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html e procure pelo tipo de dados primitivo: *Boolean, Byte, Short, Character, Integer, Long, Float, Double*.

- Para *enfileirar* (`enqueue`) elementos em nossa fila, fazemos:

```
f.add(2);
f.add(6);
f.add(8);
```

- Para saber qual elemento está na *cabeça* (`head`) de nossa fila, fazemos:

```
f.peek();
```

Como resultado dessa execução, obteremos o valor `2`.

- Para *desenfileirar* (`dequeue`) um elemento da *cabeça* de nossa fila, fazemos:

```
f.poll();
```

A execução desse código nos retorna o valor `2`. Assim, a *fila* possui agora somente os valores `6` e `8`.

- Para saber a *quantidade de elementos* de nossa fila, fazemos:

```
f.size();
```

Inicialmente, após os três *enqueues*, o tamanho era `3`, mas como executamos um *poll*, o tamanho atual é `2`.

- Para saber se a fila está *vazia*, fazemos:

```
f.isEmpty();
```

Como sabemos que seu tamanho atual é `2`, o resultado dessa execução é `false`.

Aqui exploramos apenas a *fila clássica*, mas sabemos que existem as *filas circulares*, *filas de prioridade* e *filas com dois finais (deque)*. A primeira é basicamente a transcrição dos códigos em C do capítulo 5, *Fila*, para Java. Para as filas de prioridade, Java disponibiliza a classe `PriorityQueue` e, para as filas com dois finais, a classe `ArrayDeque`.

- Para saber mais sobre a classe `PriorityQueue`, acesse: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/PriorityQueue.html.
- Para saber mais sobre a classe `ArrayDeque`, acesse: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/ArrayDeque.html.

11.4 Lista

A classe *ArrayList* é a implementação da ED *lista* em Java. Ela é a classe mais usada em Java quando se deseja usar essa ED, e também posso dizer que é a ED mais conhecida. Entretanto, ela de fato não é uma lista. Na verdade, a classe *ArrayList* é uma abstração de um *vetor*, a qual nos faz manipulá-lo como uma lista. A implementação real de uma lista como a fizemos em C no Capítulo 6 é a classe *LinkedList*, mas o uso de *ArrayList* é tão difundido que o exploraremos aqui.

- Para *criar* uma lista:

```
ArrayList<String> l = new ArrayList<>();
```

Esse código cria uma `ArrayList` (lista) chamada `l`, que armazena textos, como `Casa`, `Carro` etc. Mais uma vez, usamos *generics*.

- Para *adicionar elementos* em nossa lista, fazemos:

```
l.add("Fla");
l.add("men");
l.add("go");
```

- Para *obter um elemento* de nossa lista, fazemos:

```
l.get(0);
```

A operação `get()` recebe como parâmetro a posição do elemento dentro da estrutura. Vale lembrar que, em Java, assim como em C, as posições são *zero-based*. Nesse caso, `get(0)` nos retorna o valor `Fla`, que é o elemento da primeira posição.

- Para *excluir um elemento* de nossa lista, fazemos:

```
l.remove(1);
l.remove(1);
```

Inicialmente, com os três *adds*, eram `3` elementos, mas como executamos dois *removes*, agora temos apenas o elemento `Fla`. É válido ressaltar que, após o primeiro `remove`, a quantidade de elementos foi alterada e, consequentemente, os índices da lista. Devido a isso, o segundo `remove` é com o parâmetro `1` novamente, e não com `2`, como talvez fosse esperado.

- Para saber a *quantidade de elementos* de nossa lista, fazemos:

```
l.size();
```

Após os dois *removes*, o tamanho atual é `1`.

- Para saber se a lista está *vazia*, fazemos:

```
l.isEmpty();
```

Como sabemos que seu tamanho atual é `1`, o resultado dessa execução é `false`.

Para finalizar, é válido informar que Java não disponibiliza uma implementação nativa para *lista duplamente encadeada*. Aconselho você a implementá-la como estudo, com base nos códigos em C do capítulo 6, *Lista*.

11.5 Árvore

Java não provê uma implementação nativa de *árvore*. Podemos pensar que a classe `JTree` é responsável por isso, mas não. Ela até implementa uma árvore, mas é para ser usada em um componente visual da biblioteca do `Swing/SWT`, que possui componentes para construirmos aplicações Desktop, como o velho e conhecido Delphi. Ou seja, essa classe realmente não é o que precisamos.

Devido a isso, vamos ter aqui uma implementação própria de uma *árvore*. Abordaremos a *árvore binária ordenada*. Assim, teremos mais uma implementação, já que no Capítulo 8, *Árvore*, implementamos os tipos *clássica*, *AVL* e *N-ária*. Como sabemos, Java é uma linguagem Orientada a Objetos (OO), então é bom conhecer um pouco de OO para entender os códigos a seguir. Mais uma vez, aconselho a leitura de *Orientação a Objetos: Aprenda seus conceitos e suas aplicabilidades de forma efetiva* (ISBN: 978-85-5519-213-5).

Vamos definir a estrutura da árvore, bem como suas operações:

```
public class No {

    private int dado;
```

```
private No filhoEsquerdo;
private No filhoDireito;

public No(int valor) {

    dado = valor;
    filhoEsquerdo = null;
    filhoDireito = null;
}

public int getDado() {
    return dado;
}

public void setDado(int dado) {
    this.dado = dado;
}

public No getFilhoEsquerdo() {
    return filhoEsquerdo;
}

public void setFilhoEsquerdo(No filhoEsquerdo) {
    this.filhoEsquerdo = filhoEsquerdo;
}

public No getFilhoDireito() {
    return filhoDireito;
}

public void setFilhoDireito(No filhoDireito) {
    this.filhoDireito = filhoDireito;
}

public boolean insert(int valor) {

    if (valor == dado) {
        return false;
    } else if (valor < dado) {
```

```
            if (filhoEsquerdo == null) {
                filhoEsquerdo = new No(valor);
                return true;
            } else {
                return filhoEsquerdo.insert(valor);
            }
        } else if (valor > dado) {
            if (filhoDireito == null) {
                filhoDireito = new No(valor);
                return true;
            } else {
                return filhoDireito.insert(valor);
            }
        }

        return false;
    }

    public No delete(int valor, No pai) {

        if (valor == dado) {
            if (filhoEsquerdo == null && filhoDireito == null) {
                if (this == pai.filhoEsquerdo) {
                    pai.filhoEsquerdo = null;
                } else {
                    pai.filhoDireito = null;
                    return this;
                }
            } else if (filhoEsquerdo != null && filhoDireito == null) {
                if (this == pai.filhoEsquerdo) {
                    pai.filhoEsquerdo = filhoEsquerdo;
                } else {
                    pai.filhoDireito = filhoEsquerdo;
                    return this;
                }
            } else if (filhoEsquerdo == null && filhoDireito != null) {
                if (this == pai.filhoEsquerdo) {
                    pai.filhoEsquerdo = filhoDireito;
                } else {
```

```
                pai.filhoDireito = filhoDireito;
                return this;
            }
        } else if (filhoEsquerdo != null && filhoDireito != null) {
            dado = filhoDireito.menorValor();
            return filhoDireito.delete(dado, this);
        }
    } else if (valor < dado) {
        if (filhoEsquerdo != null) {
            return filhoEsquerdo.delete(valor, this);
        } else {
            return null;
        }
    } else if (valor > dado) {
        if (filhoDireito != null) {
            return filhoDireito.delete(valor, this);
        } else {
            return null;
        }
    }
    return null;
}

public int menorValor() {

    if (filhoEsquerdo == null)
        return dado;
    else
        return filhoEsquerdo.menorValor();
}

public boolean search(int valor) {

    if (valor == dado) {
        return true;
    } else if (valor < dado) {
        if (filhoEsquerdo == null) {
            return false;
        } else {
```

```
            return filhoEsquerdo.search(valor);
        }
    } else if (valor > dado) {
        if (filhoDireito == null) {
            return false;
        } else {
            return filhoDireito.search(valor);
        }
    }
    return false;
}

public void exibirPreOrdem() {

    System.out.print(dado + " ");

    if (filhoEsquerdo != null) {
        filhoEsquerdo.exibirPreOrdem();
    }

    if (filhoDireito != null) {
        filhoDireito.exibirPreOrdem();
    }
}

public void exibirEmOrdem() {

    if (filhoEsquerdo != null) {
        filhoEsquerdo.exibirEmOrdem();
    }

    System.out.print(dado + " ");

    if (filhoDireito != null) {
        filhoDireito.exibirEmOrdem();
    }
}

public void exibirPosOrdem() {
```

```
        if (filhoEsquerdo != null) {
            filhoEsquerdo.exibirPosOrdem();
        }

        if (filhoDireito != null) {
            filhoDireito.exibirPosOrdem();
        }

        System.out.print(dado + " ");
    }

}
```

O código anterior é responsável por definir as informações que um *nó* possui: `private int dado;`, `private No filhoEsquerdo;` e `private No filhoDireito;`. Essa classe também define as operações que o *nó* (e, consequentemente, a árvore) fornece: `insert`, `delete` e `search`. Adicionalmente, temos as três formas de navegação, `exibirPreOrdem()`, `exibirEmOrdem()` e `exibirPosOrdem()`.

Seria enfadonho explorar aqui esse código ao máximo, pois já explicamos como uma árvore binaria ordenada se comporta no capítulo 8. Então, revise e acompanhe a execução do código disponibilizado.

11.6 Set

Java provê três classes para a ED *set*: *HashSet*, *LinkedHashSet* e *TreeSet*. Existem pequenas diferenças entre elas, que são:

- **HashSet**: não mantém a ordem dos elementos inseridos;
- **TreeSet**: ordena naturalmente os elementos inseridos, tipo ascendente para números e alfabética para textos;
- **LinkedHashSet**: mantém a ordem dos elementos inseridos.

Vamos focar na mais usada e conhecida: *HashSet*.

- Para *criar* um *set*, fazemos:

```
HashSet<Integer> s = new HashSet<>();
```

No código anterior, criamos um `HashSet` (*set*) chamado `s` que armazena números (`Integer`), como `18` , `95` etc.

- Para *adicionar elementos* ao nosso *set*, fazemos:

```
s.add(1980);
s.add(1981);
s.add(1982);
s.add(1983);
s.add(1981);
s.add(2009);
```

Vale relembrar que, como essa ED é a implementação de conjuntos (conceito matemático), a segunda tentativa de adicionar o `1981` (`s.add(1981);`) não surte efeito, ou seja, ele não é adicionado no *set*, pois esse valor já tinha sido adicionado anteriormente.

- Para saber se um elemento *está contido no set*, fazemos:

```
s.contains(1992);
```

Como resultado dessa execução, obteremos o valor `false` , pois `1992` não é um valor pertencente ao nosso *set*.

- Para *remover um elemento* de nosso *set*, fazemos:

```
s.remove(2009);
```

A execução desse código nos retorna o valor `true` , indicando que o elemento `2009` foi removido. Caso o elemento passado como parâmetro não exista no *set*, `false` é retornado.

- Para saber *a quantidade de elementos* de nosso *set*, fazemos:

```
s.size();
```

Inicialmente, após os seis *adds*, o tamanho era `5`, mas como executamos um *remove*, o tamanho atual é `4`.

- Para saber se o *set* está *vazio*, fazemos:

```
s.isEmpty();
```

Como sabemos que seu tamanho atual é `4`, o resultado dessa execução é `false`.

Sets são estruturas que não garantem ordem, ou seja, os elementos estão dispersos e desordenados nessa ED; assim, não temos uma forma direta de acessá-los, tipo um `get`. Entretanto, essa ED disponibiliza um *iterator* para podermos navegar pelos seus elementos, o que veremos a seguir:

```
for (Iterator<Integer> iterator = s.iterator(); iterator.hasNext();) {
  Integer elemento = iterator.next();
  System.out.println(elemento);
}
```

Um resultado dessa execução pode ser: `1980`, `1981`, `1982` e `1983`. Quando é dito "um resultado dessa execução pode ser" é porque *sets* não garantem ordem. Se mais de uma iteração sobre ele for feita, o resultado será diferente. Com *sets* possuindo centenas ou milhares de elementos, esse comportamento se torna mais evidente. Nesse nosso simples *set* com quatro elementos talvez isso não ocorra.

Como dito anteriormente, aqui exploramos apenas a classe `HashSet`. Para saber mais sobre *LinkedHashSet* e *TreeSet*, você pode acessar os seguintes links:

- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/LinkedHashSet.html (Classe LinkedHashSet)
- https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/TreeSet.html (Classe TreeSet)

Caso queira também saber um pouco mais sobre o *iterator*, acesse: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Iterator.html.

11.7 Tabela de dispersão (Hashtable)

Duas são as classes para essa ED: *HashMap* e *Hashtable*. A segunda é mais antiga que a primeira e foi por muito tempo utilizada, mas a documentação de Java atual aconselha a usar *HashMap* por ser mais nova e solucionar problemas de performance e comportamento em relação à *Hashtable*. Devido a isso, vamos abordar *HashMap*. Caso precise saber um pouco sobre *Hashtable* para manter códigos mais antigos, acesse: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Hashtable.html.

- Para *criar* um *HashMap*, fazemos:

```
HashMap<String,Integer> m = new HashMap<>();
```

Criamos um `HashMap` chamado `m`, o qual armazena números (`Integer`) e possui como chave textos (`String`).

- Para *adicionar elementos* ao nosso *HashMap*, fazemos:

```
m.put("Brasil",5);
m.put("Alemanha",4);
m.put("Itália",4);
m.put("Argentina",2);
m.put("França",2);
m.put("Uruguai",2);
m.put("Inglaterra",1);
m.put("Espanha",1);
```

Vale relembrar que, como essa ED não permite a repetição de chaves — embora chaves diferentes possam resultar em um mesmo

valor *hash* —, se uma chave repetida for passada acidentalmente, isso ocasionará na atualização do valor correspondente à chave e não em uma nova inserção, que resultaria em uma duplicação. Ou seja, o valor original será perdido.

- Para saber se *uma chave está contida* no *HashMap* fazemos:

```
m.containsKey("Brasil");
```

Como resultado dessa execução, obteremos o valor `true`, pois "Brasil" é uma chave pertencente ao nosso *HashMap*.

- Para saber se *um valor está contido* no *HashMap* fazemos:

```
m.containsValue(10);
```

Como resultado dessa execução, obteremos o valor `false`, pois nosso *HashMap* só guarda campeões mundiais e nenhum deles foi 10 vezes campeão.

- Para *obter um valor* do nosso *HashMap*, fazemos:

```
m.get("Alemanha");
```

Como resultado dessa execução, obteremos `4`, pois esse é o valor associado à chave `Alemanha`. Se a chave passada como parâmetro não estiver contida no *HashMap*, `null` é retornado.

- Para *alterar um valor* do nosso *HashMap*, fazemos:

```
m.put("Argentina", 3);
```

Como a chave `Argentina` já existe no *HashMap*, seu valor é modificado para `3` e o valor antigo é retornado, no caso `2`. Caso não existisse, a inclusão seria feita e `null` seria retornado. Alternativamente, existe a operação `replace`, que de fato faz a modificação e retorna `true` em caso de sucesso. Caso não a chave não exista, o valor não é alterado e retorna `false`. Entretanto, essa operação é praticamente não utilizada.

- Para *remover um elemento* de nosso *HashMap*, fazemos:

```
m.remove("Argentina");
```

A execução desse código nos retorna o valor `3`, que é o valor associado à chave `Argentina`. Isso indica que a remoção foi efetivada. Caso a chave passada como parâmetro não exista, `null` é retornado.

- Para saber a *quantidade de elementos*, fazemos:

```
m.size();
```

Inicialmente, após os oito *puts*, o tamanho era `8`, mas como executamos um *remove*, o tamanho atual é `7`.

- Para saber se o *set* está *vazio*, fazemos:

```
m.empty();
```

Como sabemos que seu tamanho atual é `7`, o resultado dessa execução é `false`.

Por fim, assim como *sets*, *HashMaps* são estruturas que não garantem ordem. Mais uma vez, podemos usar um *iterator* para navegarmos pelos seus elementos. Porém, para utilizarmos ele, temos que escolher se o faremos pelos valores ou pelas chaves. A seguir, vemos isso para ambos os casos:

```
//Pelos valores
for (Iterator<Integer> i = m.values().iterator(); i.hasNext();) {
  Integer valor = i.next();
  System.out.println(valor);
}

//Pelas chaves
for (Iterator<String> i = m.keySet().iterator(); i.hasNext();) {
  String chave = i.next();
  System.out.println(chave);
}
```

Mais uma vez, possíveis resultados destas execuções podem ser, respectivamente: `5, 1, 2, 2, 1, 4, 4`; e `Brasil, Inglaterra, França, Uruguai, Espanha, Itália, Alemanha`.

Para finalizar o estudo de *HashMaps*, ele disponibiliza um método muito útil chamado *merge*, que provê um comportamento alternativo ao método *put*. Tínhamos visto que este inseria ou mudava completamente um valor associado a uma chave. Já *merge*, como o nome sugere, mescla possíveis chaves já existentes ou somente adiciona em caso de não existência. Para exemplificar, vejamos o código a seguir:

```
HashMap<String,Integer> m = new HashMap<>();
m.put("A", 1);
m.put("B", 2);

m.merge("C", 3, (valorVelho, valorNovo) -> valorVelho + valorNovo);
m.merge("B", 3, (valorVelho, valorNovo) -> valorVelho + valorNovo);
```

Após a execução desse código o *HashMap m* conterá os valores `1`, `5` e `3`, pois como a chave `B` já existia foi feita a mesclagem, no caso, a soma de `2` e `3`. Já a chave `C` não existia, então esta somente foi acrescentada com seu valor. Mais uma vez, o retorno é o novo valor ou `null` caso seja uma chave nova para o *HashMap*.

Para saber mais...

Caso precise de mais informações sobre estruturas de dados em Java, aconselho a leitura dos tutoriais abaixo. Eles foram a base para este capítulo.

- https://docs.oracle.com/javase/tutorial/collections/index.html
- https://docs.oracle.com/javase/tutorial/collections/implementations/index.html
- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

Outro conteúdo de grande relevância no universo das ED em Java é *Streams*. Embora não seja uma ED em si, é uma API do Java muito útil para manipular algumas de suas EDs, principalmente listas. Caso seja necessário, acesse os links a seguir para um conteúdo mais completo sobre esse assunto.

- https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html
- https://www.oracle.com/technical-resources/articles/java/architect-streams-pt2.html

Capítulo 12
Estruturas de Dados em C#

12.1 Vetor e Matriz

Embora C# disponibilize a classe *Array* para trabalharmos com *vetores*, vamos explorá-los de forma nativa, assim como faremos com as *matrizes*.

> O código completo está disponível no link: https://github.com/thiagoleiteecarvalho/ed-CSharp.git. Clone-o para um melhor estudo.

- Para *criar* um vetor e uma matriz, fazemos:

```
int[] v = new int[5];
int[,] m = new int[5,5];
```

No código anterior, criamos um vetor chamado `v` e uma matriz chamada `m`. Ambos são estruturas que armazenam inteiros (`int`). Essas duas definições são as mais usuais. Outras opções de criar e já inicializar vetores e matrizes são:

```
int[] v1 = {10,25};
int[] v2 = new int[] {1,2,3};
int[,] m1 = new int[4,2] {{1,2},{3,4},{5,6},{7,8}};
int[,] m2 = new int[,] {{1,2},{3,4},{5,6},{7,8}};
int[,,] m3 = new int[,,] {{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}};
int[,,] m4 = new int[2,2,3] {{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}};
int[,] m5 = {{1,2},{3,4},{5,6},{7,8}};
```

Vale ressaltar que, nas opções anteriores, quando os valores iniciais já são definidos na criação do vetor e da matriz, informar os tamanhos se torna opcional.

- Para *colocar elementos* em nosso vetor ou matriz, fazemos:

```
v[0] = 70;
v[3] = 45;
m[0,0]= 65;
m[3,2]= 17;
```

No código anterior, vemos que precisamos fornecer a posição dentro da estrutura cujo valor desejamos armazenar. Lembremos que C# é *zero-based*.

- Para *acessar elementos* em nosso vetor ou matriz, fazemos:

```
int iV = v[0];
int iM = m[0,0];
```

No código anterior, vemos que a forma de acessar os valores é bem parecida com a de armazená-los. Assim, `iV` e `iM` recebem os valores `70` e `65`, respectivamente.

- Para saber a *quantidade de elementos* de nossos vetores e matrizes, fazemos:

```
v.Length;
m.Length;
```

O vetor `v` foi criado com cinco posições, então `v.Length;` tem como resultado `5`. A matriz `m` foi definida com `[5,5]`. Nesse caso, o resultado de `m.Length` é `25`.

Para finalizar, em C#, nos *vetores* e *matrizes* não existe a ideia de excluir elementos. Isso vem do fato de estas EDs sempre serem uma estrutura de tamanho fixo. O máximo que se faz é atualizar a posição para um valor que não tenha significado para o problema em resolução.

12.2 Pilha

A classe `Stack` é a implementação da ED *pilha* em C#. Obviamente, a classe possui esse nome porque "pilha" em inglês é "*stack*". Vejamos a seguir como manipular essa classe em C#.

- Para criar uma *pilha*:

```
Stack<string> p = new Stack<string>();
```

No código anterior, criamos uma `Stack` (pilha) chamada `p`, que armazena caracteres (`string`), como "A", "B", "CD" etc.

> Vale ressaltar que usamos *generics* para especificar o *tipo de dado* da nossa pilha. Dessa forma, o `<>` é responsável por "generalizá-la". Para aprender mais sobre *generics*, acesse https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics.

- Para *empilhar* (`push`) elementos em nossa pilha, fazemos:

```
p.Push("F");
p.Push("L");
p.Push("A");
```

- Para saber qual elemento está no *topo* (`top`) de nossa pilha, fazemos:

```
p.Peek();
```

Como resultado dessa execução, obteremos o valor `A`.

- Para *remover* (`pop`) um elemento de nossa pilha, fazemos:

```
p.Pop();
```

A execução desse código nos retorna o valor `A`. Assim, a pilha possuirá agora somente os valores `F` e `L`.

- Para saber *a quantidade de elementos* de nossa pilha, fazemos:

```
p.Count;
```

Inicialmente, após os três *pushes*, o tamanho era `3`, mas como executamos um *pop*, o tamanho atual é `2`.

Adicionalmente, essa classe em C# fornece uma operação chamada `Contains`, que *procura um elemento* dentro da pilha. Para usá-la, fazemos:

```
p.Contains("F");
```

O resultado dessa execução é `true`, pois esse foi o primeiro elemento empilhado na *pilha*. Alternativamente, podemos usar essa operação para determinar se a *pilha* está vazia (*empty*) ou não. Para resultados iguais a `0`, podemos aferir que está vazia; para valores maiores ou iguais a `1`, não está vazia.

12.3 Fila

C# provê a classe `Queue` como implementação da ED *fila*.

- Para *criar* uma fila em C#:

```
Queue<Int32> f = new Queue<Int32>();
```

Esse código cria uma `Queue` (fila) chamada `f`, que armazena inteiros (`Int32`), como `1`, `2` etc. Novamente, usamos *generics*.

> `Int32` é uma *struct* que representa o tipo de dados primitivo `int`. Além desta, existem outras *structs* em C#. Para aprender mais, procure por *structs* para `int`, `double` e `bool`.

- Para *enfileirar* (`enqueue`) elementos em nossa fila, fazemos:

```
f.Enqueue(2);
f.Enqueue(6);
```

```
f.Enqueue(8);
```

- Para saber qual elemento está na *cabeça* (`head`) de nossa fila, fazemos:

```
f.Peek();
```

Como resultado dessa execução, obteremos o valor `2` .

- Para *desenfileirar* (`dequeue`) um elemento da *cabeça* de nossa fila, fazemos:

```
f.Dequeue();
```

A execução desse código nos retorna o valor `2` . Assim, a fila possui agora somente os valores `6` e `8` .

- Para saber a *quantidade de elementos* de nossa fila, fazemos:

```
f.Count;
```

Após os três `Enqueue` , o tamanho era `3` , mas após um `Dequeue` , o tamanho atual é `2` . A mesma ideia do `Count` para *empty* explicada na seção *pilha* pode ser aplicada aqui.

- Adicionalmente, essa classe em C# fornece uma operação chamada `Contains` , que *procura um elemento* dentro da fila. Para usá-la, fazemos:

```
f.Contains(10);
```

O resultado dessa execução é `false` , pois esse elemento não está contido na *fila*.

Exploramos aqui apenas a *fila clássica*, mas sabemos que existem as *filas circulares*, as *filas de prioridade* e as *filas com dois finais* (*deque*). A primeira é basicamente a transcrição para C# dos códigos em C do capítulo 5, *Fila*. Para as filas de prioridade, temos a classe `PriorityQueue` e, para as filas com dois finais, a classe `LinkedList` .

- Para saber mais sobre a classe `PriorityQueue`, acesse: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.priorityqueue-2?view=net-7.0.
- Para saber mais sobre a classe `LinkedList`, acesse: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?redirectedfrom=MSDN&view=net-7.0.

12.4 Lista

A classe `List` é a implementação da ED *lista* em C#.

- Para *criar* uma lista em C#:

```
List<string> l = new List<string>();
```

O código anterior cria uma `List` (lista) chamada `l`, a qual armazena textos, como `Cama`, `Travesseiro` etc. Mais uma vez, usamos *generics*.

- Para *adicionar elementos* em nossa lista, fazemos:

```
l.Add("Fla");
l.Add("men");
l.Add("go");
```

- Para *obter um elemento* de nossa lista, fazemos:

```
l[0];
```

Podemos notar que a forma de acesso a um elemento de uma *lista* em C# é igual à de um *vetor*, com o uso de `[]`, e não através de uma operação como nas outras estruturas. Logo, `l[0]` nos retorna o valor `Fla`, que é o elemento da primeira posição.

- Para *excluir um elemento* de nossa lista, fazemos:

```
l.RemoveAt(1);
l.RemoveAt(1);
```

Inicialmente, após os três *adds*, eram `3` elementos, mas como executamos dois *removes*, agora temos apenas o elemento `Fla`. É válido ressaltar que após o primeiro `RemoveAt` a quantidade de elementos foi alterada e, consequentemente, os índices da lista. Devido a isso, o segundo `RemoveAt` é com o parâmetro `1` novamente e não `2`, como talvez fosse esperado.

- Para saber a *quantidade de elementos* de nossa lista, fazemos:

```
l.Count;
```

Após os dois *removes*, o tamanho atual é `1`. Novamente, `Count` para emular *empty* pode ser usado.

Para finalizar, é válido informar que C# não disponibiliza uma implementação nativa para *lista duplamente encadeada*. Aconselho você a implementá-la como estudo, baseando-se nos códigos em C do capítulo 6, *Lista*.

12.5 Árvore

C# não provê uma implementação nativa de *árvore*. Existe uma classe chamada `TreeView`, mas ela não tem relação com a ED *árvore*. Ela até manipula uma árvore, mas é para ser usada em um componente visual da biblioteca do `Forms`, o qual possui componentes para construirmos aplicações Desktop, como o velho e conhecido Delphi. Ou seja, essa classe realmente não é o que precisamos.

Devido a isso, vamos ter aqui uma implementação própria de uma árvore. Abordaremos a *árvore binária ordenada* e, dessa forma, teremos mais uma implementação, já que no capítulo 8, *Árvore*, implementamos os tipos *clássica*, *AVL* e *N-ária*. Como sabemos, C#

é uma linguagem Orientada a Objetos (OO), então é bom conhecer um pouco de OO para entender os códigos a seguir. Novamente, aconselho a leitura de *Orientação a Objetos: Aprenda seus conceitos e suas aplicabilidades de forma efetiva* (ISBN: 978-85-5519-213-5).

Assim, vamos definir a estrutura da *árvore*, bem como suas operações:

```
public class No
{

    int dado;
    No filhoEsquerdo;
    No filhoDireito;

    public No(int valor)
    {
        dado = valor;
        filhoEsquerdo = null;
        filhoDireito = null;
    }

    public int Dado
    {
        get { return dado; }
        set { dado = value; }
    }

    public No FilhoEsquerdo
    {
        get { return filhoEsquerdo; }
        set { filhoEsquerdo = value; }
    }

    public No FilhoDireito
    {
        get { return filhoDireito; }
        set { filhoDireito = value; }
    }
```

```
public bool Insert(int valor)
{
    if (valor == dado)
    {
        return false;
    }
    else if (valor < dado)
    {
        if (filhoEsquerdo == null)
        {
            filhoEsquerdo = new No(valor);
            return true;
        }
        else
        {
            return filhoEsquerdo.Insert(valor);
        }
    }
    else if (valor > dado)
    {
        if (filhoDireito == null)
        {
            filhoDireito = new No(valor);
            return true;
        }
        else
        {
            return filhoDireito.Insert(valor);
        }
    }
    return false;
}

public No Delete(int valor, No pai)
{
    if (valor == dado)
    {
        if (filhoEsquerdo == null && filhoDireito == null)
```

```
{
    if (this == pai.filhoEsquerdo)
    {
        pai.filhoEsquerdo = null;
    }
    else
    {
        pai.filhoDireito = null;
        return this;
    }
}
else if (filhoEsquerdo != null && filhoDireito == null)
{
    if (this == pai.filhoEsquerdo)
    {
        pai.filhoEsquerdo = filhoEsquerdo;
    }
    else
    {
        pai.filhoDireito = filhoEsquerdo;
        return this;
    }
}
else if (filhoEsquerdo == null && filhoDireito != null)
{
    if (this == pai.filhoEsquerdo)
    {
        pai.filhoEsquerdo = filhoDireito;
    }
    else
    {
        pai.filhoDireito = filhoDireito;
        return this;
    }
}
else if (filhoEsquerdo != null && filhoDireito != null)
{
    dado = filhoDireito.MenorValor();
    return filhoDireito.Delete(dado, this);
```

```
        }
    }
    else if (valor < dado)
    {
        if (filhoEsquerdo != null)
        {
            return filhoEsquerdo.Delete(valor, this);
        }
        else
        {
            return null;
        }
    }
    else if (valor > dado)
    {
        if (filhoDireito != null)
        {
            return filhoDireito.Delete(valor, this);
        }
        else
        {
            return null;
        }
    }
    return null;
}

public int MenorValor()
{
    if (filhoEsquerdo == null)
    {
        return dado;
    }
    else
    {
        return filhoEsquerdo.MenorValor();
    }
}
```

```
public bool Search(int valor)
{
    if (valor == dado)
    {
        return true;
    }
    else if (valor < dado)
    {
        if (filhoEsquerdo == null)
        {
            return false;
        }
        else
        {
            return filhoEsquerdo.Search(valor);
        }
    }
    else if (valor > dado)
    {
        if (filhoDireito == null)
        {
            return false;
        }
        else
        {
            return filhoDireito.Search(valor);
        }
    }
    return false;
}

public void ExibirPreOrdem()
{

    Console.WriteLine(dado + " ");

    if (filhoEsquerdo != null)
    {
        filhoEsquerdo.ExibirPreOrdem();
```

```
    }

    if (filhoDireito != null)
    {
        filhoDireito.ExibirPreOrdem();
    }
}

public void ExibirEmOrdem()
{

    if (filhoEsquerdo != null)
    {
        filhoEsquerdo.ExibirEmOrdem();
    }

    Console.WriteLine(dado + " ");

    if (filhoDireito != null)
    {
        filhoDireito.ExibirEmOrdem();
    }
}

public void ExibirPosOrdem()
{

    if (filhoEsquerdo != null)
    {
        filhoEsquerdo.ExibirPosOrdem();
    }

    if (filhoDireito != null)
    {
        filhoDireito.ExibirPosOrdem();
    }

    Console.WriteLine(dado + " ");
}
```

```
}
```

O código anterior é responsável por definir as informações que um *nó* possui: `int dado;`, `No filhoEsquerdo;` e `No filhoDireito;`. Essa classe também define as operações que o *nó* (e, consequentemente, a *árvore*) fornece: `Insert`, `Delete` e `Search`. Adicionalmente, temos as três formas de navegação: `ExibirPreOrdem()`, `ExibirEmOrdem()` e `ExibirPosOrdem()`. Não vamos explorar aqui ao máximo esse código, pois seria muito enfadonho. Além disso, já explicamos como uma *árvore binária ordenada* se comporta no capítulo 8. Então, revise e acompanhe a execução do código disponibilizado.

12.6 Set

C# provê duas classes para a ED *set*: `HashSet` e `SortedSet`. Existe apenas uma diferença entre elas: `HashSet` não garante ordem e `SortedSet` garante. Vamos aqui explorar a `HashSet`, por ser a mais conhecida e utilizada.

- Para *criar* um *set*, fazemos:

```
HashSet<string> s = new HashSet<string>();
```

No código anterior, criamos um `HashSet` (*set*) chamado `s` que armazena textos (`strings`), como "A", "B" etc.

- Para *adicionar elementos* ao nosso *set*, fazemos:

```
s.Add("1980");
s.Add("1981");
s.Add("1982");
s.Add("1983");
s.Add("1981");
s.Add("2009");
```

Como essa ED é a implementação de conjuntos (conceito matemático), a segunda tentativa de adicionar o `1981` (`s.Add("1981");`) não surte efeito, ou seja, `1981` não é adicionado no *set*, pois esse valor já tinha sido adicionado anteriormente.

- Para saber se um elemento *está contido no set*, fazemos:

```
s.Contains("1992");
```

Como resultado dessa execução, obteremos o valor `false` , pois `1992` não é um valor pertencente ao nosso *set*.

- Para *remover um elemento* de nosso *set*, fazemos:

```
s.Remove("2009");
```

A execução desse código nos retorna o valor `true` , indicando que o elemento `2009` foi removido. Caso o elemento passado como parâmetro não existisse no *set*, `false` seria retornado.

- Para saber a *quantidade de elementos* de nosso *set*, fazemos:

```
s.Count;
```

Inicialmente, após os seis *adds*, o tamanho era `5` , mas como executamos um *remove*, o tamanho atual é `4` . Essa classe em C# não disponibiliza uma operação `empty` para verificar se o *set* está vazio. Assim, podemos usar `Count` para determinar isso. Se retornar `0` , é sinal de que o *set* está vazio; caso contrário, retornará a quantidade atual, como visto anteriormente.

- Para *iterar* sobre os elementos nessa ED em C#, fazemos:

```
foreach(var valor in s)
{
  Console.WriteLine(valor);
}
```

Um resultado dessa execução pode ser: `1980` , `1981` , `1982` e `1983` .

Como dito anteriormente, aqui exploramos apenas a classe `HashSet`. Para saber mais sobre `SortedSet`, acesse:

- https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedset-1?view=net-7.0.

12.7 Tabela de dispersão (Hashtable)

São duas as classes para essa ED: `Hashtable` e `Dictionary`. A segunda é mais utilizada do que a primeira por prover um acesso mais rápido aos elementos, devido a usar *generics* e eliminar a necessidade de *boxing/unboxing*. Assim, vamos explorar aqui a classe `Dictionary`.

- Para *criar* um `Dictionary`, fazemos:

```
Dictionary<string, int> d = new Dictionary<string, int>();
```

Nesse código, criamos um `Dictionary` chamado `d`, o qual armazena números (`int`) e possui como chave textos (`string`).

- Para *adicionar elementos* ao nosso *dictionary*, fazemos:

```
d.Add("Brasil",5);
d.Add("Alemanha",4);
d.Add("Itália",4);
d.Add("Argentina",2);
d.Add("França",2);
d.Add("Uruguai",2);
d.Add("Inglaterra",1);
d.Add("Espanha",1);
```

Vale relembrar que, como essa ED não permite a repetição de chaves — embora chaves diferentes possam resultar em um mesmo valor *hash* —, se uma chave repetida for passada acidentalmente, isso ocasionará a atualização do valor correspondente à chave e não

uma nova inserção, o que resultaria em uma duplicação. Ou seja, o valor original será perdido.

- Para saber se uma *chave* está contida no *dictionary*, fazemos:

```
d.ContainsKey("Brasil");
```

Como resultado dessa execução, obteremos o valor `true`, pois `Brasil` é uma chave pertencente ao nosso *dictionary*.

- Para saber se um *valor* está contido no *dictionary*, fazemos:

```
d.ContainsValue(10);
```

Como resultado dessa execução, obteremos o valor `false`, pois nosso `Dictionary` só guarda campeões mundiais e nenhum deles foi 10 vezes campeão.

- Para *obter um valor* do nosso *dictionary*, fazemos:

```
d["Alemanha"];
```

Como resultado dessa execução, obteremos `4`, pois esse é o valor associado à chave `Alemanha`. Se a chave passada como parâmetro não estiver contida no *dictionary*, `null` é retornado.

- Para *alterar um valor* do nosso *dictionary*, fazemos:

```
d["Argentina"] = 3;
```

Como a chave `Argentina` já existe no *dictionary*, seu valor é modificado para `3`. Caso não existisse, a inclusão seria feita. Ou seja, além de `Add`, temos uma segunda opção para adicionar elementos no `Dictionary`.

- Para *remover um elemento* de nosso *dictionary*, fazemos:

```
d.Remove("Argentina");
```

A execução desse código apaga a chave e o valor cuja chave for igual a `Argentina`. Além disso, retorna o valor `true`, pois essa chave

foi encontrada e retirada da estrutura. Caso a chave passada como parâmetro não exista, `false` é retornado.

- Para saber a *quantidade de elementos*, fazemos:

```
d.Count;
```

Inicialmente, após os oito *puts*, o tamanho era `8`, mas como executamos um *remove*, o tamanho atual é `7`.

Temos duas formas de percorrer nosso *dictionary*: pelas chaves e pelos valores. A seguir, vamos ver os códigos para essas duas abordagens.

```
//Pelas chaves
foreach(var valor in d.Keys)
{
  Console.WriteLine(valor);
}

//Pelos valores
foreach(var valor in d.Values)
{
  Console.WriteLine(valor);
}
```

Mais uma vez, possíveis resultados dessas execuções seriam, respectivamente: `5, 1, 2, 2, 1, 4` e `4`; e `Brasil, Inglaterra, França, Uruguai, Espanha, Itália` e `Alemanha`.

Por fim, para saber mais sobre *Hashtable*, acesse: https://learn.microsoft.com/en-us/dotnet/api/system.collections.hashtable?view=net-7.0.

Para saber mais...

Caso precise de mais informações sobre estruturas de dados em C#, aconselho a leitura dos conteúdos abaixo. Eles foram a base para este capítulo.

- https://learn.microsoft.com/en-us/dotnet/api/system.collections?view=net-7.0
- https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic?view=net-7.0

Capítulo 13
Estruturas de Dados em Python

13.1 Vetor e Matriz

Python provê uma classe chamada *Array* que auxilia na criação de *vetores*. Para *matrizes*, é aconselhado o uso da biblioteca *numpy*, a qual deve ser instalada separadamente.

> O código completo está disponível no link: https://github.com/thiagoleiteecarvalho/ed-Python.git. Clone-o para um melhor estudo.

- Para criar um *vetor* e uma *matriz*, fazemos:

```
v = array('i',[1,4,78,29])
m1 = numpy.matrix([['a','b'],['c','d']])
m2 = numpy.array([(1,2,3),(4,5,6)])
```

Nesse código, criamos um *vetor* chamado `v` de inteiros e duas *matrizes*, chamadas `m1` e `m2`, de caracteres e inteiros, respectivamente.

Explicando a definição do vetor, temos `array(data type,[value list])`, onde `data type` é a definição do *tipo de dado* que o vetor armazenará e `value list`, os *valores*, que devem ser compatíveis com o tipo de dado. Para `data type`, temos um conjunto predefinido de tipos, que estão a seguir:

Código do tipo	Tipo de dado	Tamanho em bytes
i ou I	int	2
b ou B	unicode character	2

Código do tipo	Tipo de dado	Tamanho em bytes
h ou H	int	2
l ou L	int	4
f	float	4
d	float	8

Dessa forma, `array('i',[1,4,78,29])` cria um vetor de inteiros com os elementos `1`, `4`, `78` e `29`.

Explicando a segunda e terceira definições (para matriz), que usam a biblioteca *numpy*, temos:

1) `numpy.matrix([['a','b'],['c','d']])` usa a operação `matrix` da biblioteca *numpy* para definir uma *matriz*. Com o uso dela, não precisamos prover o *tipo de dado*, pois isso é inferido a partir dos valores. Nesse caso, criamos uma matriz de *caracteres*.

2) `numpy.array([(1,2,3),(4,5,6)])` usa a operação `array` para também definir uma *matriz*. Mais uma vez, não precisamos prover o *tipo de dado*, pois isso é inferido a partir dos valores. Existe apenas uma pequena diferença na passagem dos valores iniciais: nessa, usamos parênteses e colchetes; na anterior, eram somente colchetes. O resultado final é o mesmo. Nessa declaração, criamos uma matriz de *inteiros*.

- Para *colocar elementos* em nosso vetor ou matriz, fazemos:

```
v[0] = 2019
m1[0,1] = 'F'
m2[1,1] = 2020
```

No código anterior, vemos que, para armazenar os elementos, precisamos fornecer a posição dentro da estrutura na qual desejamos armazená-lo. Devemos nos lembrar também de que Python é *zero-based*. Para `v` passamos apenas um índice, mas para

`m1` e `m2` passamos dois índices, pois uma *matriz* é um "vetor de vetores".

- Para *remover elementos* em nosso vetor, fazemos:

```
del v[0]
```

Ao contrário de muitas linguagens OO, Python permite a remoção de elementos em *vetores*. O código anterior usa a diretiva `del` para realizar essa operação. Dessa forma, `del v[0]` remove o primeiro elemento de `v`, que é `2019`. Não é possível remover elementos de uma *matriz* em python.

- Para *adicionar elementos* em nosso vetor, fazemos:

```
v.append(2022)
```

No código anterior, o valor `2022` é adicionado ao final do vetor. Ao contrário de Java e C#, em Python essa operação é permitida. Não é possível realizar *appends* em matrizes.

- Para saber a *quantidade de elementos* de nossos vetores e matrizes, fazemos:

```
len(v)
m1.shape
m2.shape
```

O vetor `v` foi criado com quatro posições e, após isso, removemos um elemento e adicionamos outro. Então `len(v)` tem como resultado `4`. Para as matrizes `m1` e `m2`, a função `shape` retorna suas dimensões. Assim, o resultado é `(2,2)` e `(2,3)`, respectivamente.

13.2 Pilha

Python tem uma classe de uso geral (*general purpose*) que podemos usar como *pilha*: `list`. Essa classe será vista em EDs mais adiante,

mas aqui a usaremos como *pilha*.

- Para *criar* uma pilha, fazemos:

```
p = []
p1 = ['FLA','MEN','GO']
```

No código anterior, criamos duas pilhas: uma vazia (`p`) e outra de caracteres (`p1`). Se desejássemos, poderíamos criar uma *pilha* de valores numéricos também. Nos exemplos a seguir, usaremos a pilha `p` .

- Para *empilhar elementos* (`push`) em nossa pilha, fazemos:

```
p.append(1981)
p.append(1982)
p.append(1983)
```

- Para saber qual elemento está no *topo* (`top`) de nossa *pilha* temos que fazer um código "na mão", pois Python não provê essa operação. Devemos fazer da seguinte maneira:

```
p[len(p)-1]
```

Como resultado dessa execução, obteremos o valor `1983` .

- Para *desempilhar um elemento* (`pop`) de nossa pilha, fazemos:

```
p.pop()
```

A execução desse código nos retorna o valor `1983` . Assim, a pilha possuiria agora somente os valores `1981` e `1982` .

- Para saber a *quantidade de elementos* de nossa pilha, fazemos:

```
len(p)
```

Inicialmente, após os três *pushes*, o tamanho era `3` , mas como executamos um *pop*, o tamanho atual é `2` .

- Adicionalmente, a operação `count` indica a quantidade de vezes que determinado elemento aparece dentro da pilha. Assim, podemos usá-la como uma operação de pesquisa. Se o resultado for `1` ou maior, então o elemento pertence à pilha. Se for `0`, não pertence. Para usá-la, fazemos:

```
p.count(1981)
```

O resultado dessa execução é `1`, pois esse foi o primeiro elemento empilhado.

13.3 Fila

Embora em Python se possa usar também a classe *list* como uma *fila*, esse uso é desencorajado por questões de performance. Todavia, a classe *deque* é disponibilizada como opção de uso. Ela pode ser usada como *fila clássica* ou como *fila de duas cabeças* (*deque*). Vamos aqui explorar as duas opções.

- Para *criar* uma fila ou deque, fazemos:

```
f = deque()
d = deque()
```

No código anterior, criamos uma fila clássica vazia (`f`) e um deque vazio (`d`). Se desejássemos prover valores iniciais, bastava defini-los dentro de colchetes, separados por vírgula, e passá-los como parâmetro no momento da definição. Vejamos a seguir:

```
f = deque(['a','b','c'])
d = deque([1,2,3])
```

Assim, criamos uma fila de caracteres e um deque de inteiros.

- Para *enfileirar* (`enqueue`) elementos em nossa fila e deque, fazemos:

```
f.append('d')
d.append(4)
d.appendleft(0)
```

Em `f.append('d')`, enfileiramos ao final da fila, pois essa é uma fila clássica. Entretanto, `d` é um deque, que possibilita enfileirar no *fim* e no *início*. Assim, `d.append(4)` e `d.appendleft(0)` são executados para enfileirar à direita (fim) e à esquerda (início).

- Para saber qual elemento está nas *cabeças* (`heads`) de nossa fila e deque, fazemos:

```
f[0]
d[0]
d[len(d)-1]
```

A classe `deque` não provê nativamente uma operação que retorna as *cabeças* de uma fila clássica ou deque. Assim, fazemos "na mão" essas implementações. Desse modo, `f[0]` e `d[0]` são responsáveis por informar os elementos que estão nas cabeças de nossas fila clássica e deque. Já `d[len(d)-1]` é responsável por informar o elemento que está na cabeça à direita de nosso deque. Os resultados para `f[0]`, `d[0]` e `d[len(d)-1]` são, respectivamente, `a`, `0` e `4`.

- Para *desenfileirar* (`dequeue`) de nossa fila ou deque, fazemos:

```
f.popleft()
d.pop()
d.popleft()
```

Em `f.popleft()`, temos como resultado o valor `a`, pois estamos usando nossa fila clássica. Assim, o elemento `a` é retornado e removido da fila. Em nosso deque, a execução de `d.pop()` e `d.popleft()` remove e retorna, respectivamente, `4` e `0`, pois esses eram os elementos de suas extremidades (`4` estava à direita e `0`, à esquerda). Nesse caso, removemos das *duas cabeças*.

- Para saber a *quantidade de elementos* de nossas filas, fazemos:

```
len(f)
len(d)
```

Para `f` e `d`, a execução de `len` tem como resultado `3`.

- Adicionalmente, a operação `count` indica a quantidade de vezes que determinado elemento aparece dentro da fila. Assim, podemos usá-la como uma operação de pesquisa. Se o resultado for maior ou igual a `1`, então o elemento pertence à fila. Se for `0`, não pertence. Para usá-la, fazemos:

```
f.count('b')
d.count(3)
```

O resultado dessas execuções é `1`, pois esses elementos foram enfileirados nas nossas filas.

13.4 Lista

Novamente, usaremos a classe `list`. Porém, desta vez a usaremos para de fato criar uma *lista*.

- Para *criar* uma lista:

```
l = []
```

O código anterior cria uma `list` (*lista*) vazia chamada `l`. Como já vimos anteriormente, se desejarmos valores iniciais, os fornecemos separados por vírgula.

- Para *adicionar elementos* em nossa lista, fazemos:

```
l.append("Fla")
l.append("men")
l.append("go")
```

Os códigos anteriores são responsáveis por adicionar elementos do tipo texto em nossa lista. Foram adicionadas três sílabas.

- Para *obter um elemento* de nossa lista, fazemos:

```
l[0]
```

Podemos notar que a forma de acesso a um elemento de uma lista em Python é igual à de um vetor, pilha ou fila. Assim, `l[0]` nos retorna o valor `Fla`, que é o elemento da primeira posição.

- Para *excluir um elemento* de nossa lista, fazemos:

```
l.remove("Fla")
```

Inicialmente, após os três *adds*, o tamanho era de `3` elementos, mas como executamos um *remove*, agora temos apenas os elementos `men` e `go`. Podemos notar que o *remove* recebe como parâmetro o elemento a ser excluído e não uma posição.

- Para *adicionar elementos de forma posicional* em nossa lista, fazemos:

```
l.insert(0, "Fla")
```

Após a execução desse código, voltamos a possuir os elementos `Fla`, `men` e `go`, pois `Fla` foi adicionado na 1ª posição da lista novamente.

- Para saber a *quantidade de elementos* de nossa lista, fazemos:

```
len(l)
```

Após três *appends*, um *remove* e um *insert*, o tamanho atual é `3`.

Para finalizar, é válido informar que Python não disponibiliza uma implementação nativa para *lista duplamente encadeada*. Aconselho você a implementá-la como estudo, baseando-se nos códigos em C do capítulo 6, *Lista*.

13.5 Árvore

Python não provê uma implementação nativa de árvore. Devido a isso, vamos ter aqui uma implementação própria de uma árvore. Abordaremos a *árvore binária ordenada*, assim, teremos mais uma implementação, já que no capítulo 8, *Árvore*, implementamos os tipos clássica, AVL e N-ária. Embora Python seja uma linguagem Orientada a Objetos (OO), aqui veremos uma abordagem não OO. Caso seja necessário se aprofundar em OO, aconselho a leitura de *Orientação a Objetos: Aprenda seus conceitos e suas aplicabilidades de forma efetiva* (ISBN: 978-85-5519-213-5).

Assim sendo, vamos definir a estrutura da árvore, bem como suas operações:

```
class No:
    def __init__(self, dado: int, filho_esquerdo=None,
filho_direito=None):
        self.dado = dado
        self.filho_esquerdo = filho_esquerdo
        self.filho_direito = filho_direito


def insert(no: No, dado: int):

    if dado <= no.dado:
        if no.filho_esquerdo is None:
            no.filho_esquerdo = No(dado)
            return

        inserir(no.filho_esquerdo, dado)

    # ir para direita
    else:
        if no.filho_direito is None:
            no.filho_direito = No(dado)
            return

        inserir(no.filho_direito, dado)

def delete(no: No, dado: int) -> Optional[No]:
```

```
    if no is None or no.dado == dado:
        return None

    if dado <= no.dado:
        no.filho_esquerdo = remover(no.filho_esquerdo, dado)
        return no

    no.filho_direito = remover(no.filho_direito, dado)
    return no

def search(no: No, dado: int) -> Optional[No]:

    if no is None:
        return None

    if no.dado == dado:
        return no

    if dado <= no.dado:
        return buscar(no.filho_esquerdo, dado)

    return buscar(no.filho_direito, dado)

def imprimir_pre_ordem(no: No):

    print(no.dado + ' ')
    imprimir_pre_ordem(no.filho_esquerdo)
    imprimir_pre_ordem(no.filho_direito)

def imprimir_em_ordem(no: No):

    imprimir_pre_ordem(no.filho_esquerdo)
    print(no.dado + ' ')
    imprimir_pre_ordem(no.filho_direito)

def imprimir_pos_ordem(no: No):

    imprimir_pre_ordem(no.filho_esquerdo)
```

```
    imprimir_pre_ordem(no.filho_direito)
    print(no.dado + ' ')
```

O código anterior é responsável por definir as informações que um *nó* possui: `int dado;`, `No filho_esquerdo;` e `No filho_direito;`. Além disso, também definimos as operações às quais o nó pode ser submetido (e, consequentemente, a árvore): `insert`, `delete` e `search`. Adicionalmente, temos as três formas de navegação, `imprimir_pre_ordem`, `imprimir_em_ordem` e `imprimir_pos_ordem`. Não vamos explorar aqui ao máximo esse código, pois seria muito enfadonho, e já explicamos como uma *árvore binária ordenada* se comporta no capítulo 8. Então, revise e acompanhe a execução do código disponibilizado.

13.6 Set

Python provê uma classe chamada `set` para podermos criar e manipular essa estrutura.

- Para *criar* um *set*, fazemos:

```
s1 = {'fla','men','go'}
s = set()
```

Nos códigos anteriores, temos duas opções para criarmos um *set* chamado `s`. O primeiro código permite o uso de `{}`, mas, para isso, os valores iniciais são obrigatórios. Se eles não forem fornecidos, na verdade estaremos criando um `dict`, que é a ED *Hashtable* (a qual veremos mais adiante). Caso se deseje criar um *set* vazio, devemos usar a segunda opção. Todavia, essa opção também suporta valores iniciais caso necessário, separados também por vírgula.

- Para *adicionar elementos* ao nosso *set*, fazemos:

```
s.add(1980)
s.add(1981)
```

```
s.add(1982)
s.add(1983)
s.add(1981)
s.add(2009)
```

Como essa ED é a implementação de conjuntos (conceito matemático), a segunda tentativa de adicionar o `1981` (`s.add(1981)`) não surte efeito, ou seja, não é adicionado no *set*, pois esse valor já tinha sido adicionado anteriormente. Nesse exemplo, nosso *set* foi iniciado com números.

- Para saber se um elemento *está contido no set*, fazemos:

```
1992 in s
```

O código anterior usa a diretiva `in` para realizar essa operação. Como resultado dessa execução, obteremos o valor `false` , pois `1992` não é um valor pertencente ao nosso *set*.

- Para *remover um elemento* de nosso *set*, fazemos:

```
s.remove(2009)
s.remove(2019)
s.discard(2020)
```

As execuções desses códigos nos fornecem retornos diferentes. Em `s.remove(2009)` , o elemento `2009` é removido. Já em `s.remove(2019)` não ocorre uma remoção, pois esse elemento não está no *set* e, além disso, um erro é disparado devido a essa ausência. Por fim, `s.discard(2020)` tenta remover o elemento `2020` . Se ele existir, é retirado. Se não, nada acontece.

- Para saber a *quantidade de elementos* de nosso *set*, fazemos:

```
len(s)
```

Inicialmente, após os seis *adds*, o tamanho era `5` , mas como executamos um *remove*, o tamanho atual é `4` .

- Para *iterar sobre os elementos* nessa ED em Python, fazemos:

```
for valor in s:
    print(valor)
```

Um resultado dessa execução pode ser: `1980`, `1981`, `1982` e `1983`.

13.7 Tabela de dispersão (Hashtable)

A classe para essa ED em Python é a `dict` (*dictionary*).

- Para criar um *dictionary*, temos as seguintes opções:

```
d = dict()
d = {}
d = dict(alemanha=4, itália=4, argentina=2, França=1, uruguai=2,
inglaterra=1)
d = {'alemanha':4, 'itália':4, 'argentina':2, 'França':1, 'uruguai':2,
'inglaterra':1}
```

Nos códigos anteriores, temos quatro opções para criarmos um *dict* chamado `d`: duas para uma versão vazia e duas para uma versão com valores iniciais. Nessas últimas, fica evidente que `d` armazena números e possui como chave textos. A mais comunmente usada é a que utiliza `{}`.

- Para *adicionar elementos* ao nosso *dictionary*, fazemos:

```
d['brasil'] = 5
d['espanha'] = 1
d['França'] = 2
```

Nos códigos anteriores, temos dois comportamentos. Em `d['brasil'] = 5` e `d['espanha'] = 1`, ocorre a inclusão de um novo conjunto de chave/valor, pois eles não existiam em nosso *dictionary*. Já em `d['França'] = 2` ocorre uma atualização, pois a chave `França` já existia, portanto foi alterada para `2`.

- Para saber se um elemento *está contido* no nosso *dictionary*, fazemos:

```
'brasil' in d
```

O código anterior usa a diretiva `in` para realizar essa operação. Como resultado dessa execução, obteremos o valor `true`, pois `'brasil'` é uma chave pertencente ao nosso *dictionary*.

- Para *obter um valor* do nosso *dictionary*, fazemos:

```
d["Alemanha"]
```

Como resultado dessa execução, obteremos `4`, pois esse é o valor associado à chave `Alemanha`. Se a chave passada como parâmetro não estiver contida no *dictionary*, um erro é disparado devido a essa ausência.

- Para *alterar um valor* do nosso *dictionary*, fazemos:

```
d["Argentina"] = 3
```

Como a chave `Argentina` já existe no *dictionary*, seu valor é modificado para `3`. Caso não existisse, a inclusão seria feita.

- Para *remover um elemento* de nosso *dictionary*, fazemos:

```
del d['Argentina']
```

A execução desse código apaga a chave `Argentina` e o valor associado a ela. Se a chave passada como parâmetro não pertencer ao *dictionary*, um erro é disparado devido a essa ausência, o que inviabiliza a remoção.

- Para saber a *quantidade de elementos*, fazemos:

```
len(d)
```

Inicialmente, após os oito *puts*, o tamanho era `8`, mas como executamos um *remove*, o tamanho atual é `7`.

Temos duas formas de percorrer nosso *dictionary*: pelas chaves e pelos valores. A seguir, vamos ver essas duas abordagens.

```
#Pelas chaves
for valor in d.keys():
    print(valor)

#Pelos valores
for valor in d.values():
    print(valor)
```

Alguns possíveis resultados dessas execuções são, respectivamente: `alemanha , itália , frança , uruguai , inglaterra , brasil , espanha` ; e `4 , 4 , 1 , 2 , 1 , 5 , 1 .`

13.8 Tuple

`Tuple` é uma ED existente em Python que se comporta com uma *lista*, mas com uma peculiaridade: é imutável. Quando uma tupla é criada com um conjunto de valores inicias, eles não podem ser modificados ou excluídos. Outra característica de uso dessa ED é que ela normalmente é usada com elementos heterogêneos, ou seja, de tipos diferentes.

- Para *criar* um *tuple*, fazemos:

```
t = 'Flamengo', 2022
```

No código anterior, vemos que a declaração lembra as declarações vistas até aqui, mas com uma pequena diferença: não se usam colchetes, parênteses ou chaves. Apenas os valores são informados, os quais são um texto e um número. Isso faz Python criar um *tuple* e não um *list* ou *set*, por exemplo. Esse *tuple* que criamos tem exatamente 2 elementos.

- Para *acessar um elemento* de um *tuple*, fazemos:

```
t[1]
```

A execução do código anterior retorna `2022`.

- Para *percorrer os elementos* de um *tuple*, fazemos:

```
for valor in t:
    print(valor)
```

O resultado dessa execução é `Flamengo` e `2022`.

Como dito, a ED *tuple* em Python é imutável. Devido a isso, ela não suporta exclusões e alterações de seus elementos. Se essas operações forem realizadas, erros serão disparados. Essa ED é muito usada na área de Ciência de Dados para análises de dados.

Para saber mais...

Caso precise de mais informações sobre estruturas de dados em Python, aconselho a leitura dos conteúdos abaixo. Eles foram a base para este capítulo.

- https://docs.python.org/3/library/collections.html#
- https://docs.python.org/3/tutorial/datastructures.html#
- https://docs.python.org/3/library/stdtypes.html#

CAPÍTULO 14
Estruturas de Dados em JavaScript

14.1 Vetor e Matriz

JavaScript (ou simplesmente JS) tem um objeto (*built-in object*) de uso geral chamado ***Array***, que podemos utilizar para várias EDs. Aqui o usaremos como *vetor*, mas, nas seções seguintes, o usaremos de formas distintas.

> O código completo está disponível no link: https://github.com/thiagoleiteecarvalho/ed-JS.git. Clone-o para um melhor estudo.

- Para *criar* um *vetor*, fazemos:

```
var v1 = ["Fla", "men", "go"];
var v2 = new Array("Fla", "men", "go");
```

Nos códigos anteriores, temos duas maneiras de criar um vetor: a primeira é chamada de Notação Literal (*literal notation*) e a segunda é com o uso de um construtor. Após a execução dos dois códigos anteriores, teremos dois *vetores*: `v1` e `v2`.

- Para *criar* uma *matriz*, fazemos:

```
var m1 = [["F","l","a"], ["m","e","n"], ["g","o"]];

var m2 = new Array(3);
m2[0] = new Array("F","l","a");
m2[1] = new Array("m","e","n");
m2[2] = new Array("g","o");

var m3 = new Array(3);
m3[0] = ["F","l","a"];
```

```
m3[1] = ["m","e","n"];
m3[2] = ["g","o"];
```

Nos códigos anteriores, usamos as notações utilizadas para os vetores, mas com pequenas alterações, em três maneiras diferentes de criar matrizes: a primeira usando a Notação Literal; a segunda utilizando o construtor; e a terceira, com uma mistura de Notação Literal e construtor. Em todas, três matrizes foram criadas: `m1`, `m2` e `m3`.

Para finalizar, é válido dizer que vetores e matrizes em JS podem ser heterogêneos, mas, aqui, apenas utilizaremos dados do mesmo tipo.

- Para *adicionar elementos* em nossos vetores ou matrizes, fazemos:

```
v1[3] = "!";
v2[3] = "!";
m1[2][2] = "!";
m2[2][2] = "!";
m3[2][2] = "!";
```

Nos códigos anteriores, vemos que, para adicionar elementos, precisamos fornecer a posição dentro da estrutura cujo valor desejamos armazenar. Dessa forma, para `v`, passamos o índice `3` e, para `m1`, `m2` e `m3`, passamos dois índices, pois uma matriz é um "vetor de vetores". Devemos nos lembrar de que JS é *zero-based*.

É válido frisar que vetores e matrizes em JS podem ser redimensionáveis. Foi isso que permitiu o código `v2[3]`, pois, como JS é *zero-based*, o `v2 = new Array("Fla", "men", "go");` só disponibilizou inicialmente os índices `0`, `1` e `2`. Por fim, se desejássemos *alterar um elemento* de uma posição preexistente em nossos vetores ou matrizes, bastaria utilizar o valor do índice dessa posição, por exemplo, `v1[3]= ".";`.

- Para *remover elementos* em nosso vetor, fazemos:

```
v1.splice(3,1);
v2.splice(3,1);
```

Ao contrário de muitas linguagens OO, JS permite a remoção de elementos em *vetores*. O código anterior usa a função `splice` para realizar essa operação. Dessa forma, `splice(3,1);` remove a `!` anteriormente adicionada. Nessa função, o `3` é a posição da `!` no vetor e `1`, a quantidade de elementos a serem excluídos. Assim, excluímos exatamente a `!`.

Essa função ainda possibilita a exclusão de outras maneiras, mas aqui apenas exploraremos essa, por ser a mais simples e, consequentemente, a mais utilizada. Para mais detalhes, nas *Referências bibliográficas*, temos o link para um conteúdo mais vasto sobre essa função.

Para remover elementos de uma *matriz* em JS, também podemos usar a operação `splice`, que, entretanto, não removerá apenas um elemento, mas um "subvetor" inteiro. É como se eliminássemos uma dimensão da matriz.

- Para *acessar elementos* em nossos vetores ou matrizes, fazemos:

```
var s1 = v1[3];
var s2 = v2[3];
var s3 = m1[2][2];
var s4 = m2[2][2];
var s5 = m3[2][2];
```

Podemos notar que os códigos anteriores são similares aos do item *adicionar elementos*; entretanto, agora estamos obtendo os valores anteriormente adicionados — no caso, `!`.

- Para saber a *quantidade de elementos* em nossos vetores e matrizes, fazemos:

```
v1.length;
v2.length;
```

Os vetores `v1` e `v2` foram criados com três posições e, após isso, adicionamos e removemos um elemento; então `length` em ambos tem como resultado `3`. Para as matrizes `m1`, `m2` e `m3`, temos que ter um trabalho manual para descobrir seu tamanho total. Nesse caso, temos que obter o tamanho de cada uma de suas dimensões. Assim, para a matriz `m1`, temos: `m1[0].length + m1[1].length + m1[2].length`, o que resulta em `9`. O mesmo princípio se aplica a `m2` e `m3`.

14.2 Pilha

Vamos agora usar o objeto *Array* como uma *pilha*.

- Para *criar* uma pilha, fazemos:

```
var p1 = [];
var p2 = [1980, 1981, 1982, 1983];
```

Nos códigos anteriores, criamos duas *pilhas*: uma vazia (`p1`) e outra de números (`p2`). Se desejássemos, poderíamos criar uma pilha de caracteres também. Aqui também seria possível usar o construtor `new Array`, mas reservarei o uso dele somente para vetores e matrizes. Nos exemplos a seguir, usaremos a pilha `p2`.

- Para empilhar (`push`) elementos em nossa pilha, fazemos:

```
p2.push(1992);
p2.push(2009);
p2.push(2019);
```

- Para saber qual elemento está no *topo* (`top`) de nossa pilha, temos que fazer um código "à mão", pois JS não provê essa operação. Assim, devemos fazer da seguinte maneira:

```
p2[p2.length-1];
```

Como resultado dessa execução, obteremos o valor `2019`.

- Para *desempilhar* (`pop`) um elemento de nossa pilha, fazemos:

```
p2.pop();
```

A execução desse código nos retorna o valor `2019` . Assim, a pilha possuiria agora somente os valores `1981` , `1982` , `1983` , `1992` , e `2009` .

- Para saber a *quantidade de elementos* de nossa pilha, fazemos:

```
p2.length;
```

Inicialmente, criamos a *pilha* com quatro elementos. Após isso, fizemos três *pushes* e executamos um *pop*. Dessa forma, o tamanho atual é `6` .

- Adicionalmente, a operação `indexOf` indica a posição de um determinado elemento dentro da pilha. Podemos, então, usá-la como uma *operação de pesquisa*. Se o resultado for maior que `1` , então o elemento pertence à *pilha* e se encontra na posição retornada. Se for `-1` , não pertence. Para usá-la, fazemos:

```
p2.indexOf(1981);
p2.indexOf(2020);
```

O resultado da primeira execução é `1` , pois esse foi o segundo elemento empilhado na pilha. Já para `2020` é `-1` , pois esse ano não foi empilhado em nossa pilha.

14.3 Fila

Agora vamos usar o objeto *Array* como uma *fila*.

- Para *criar* uma fila, fazemos:

```
var f = ["FLA","MEN","GO"];
```

No código anterior, criamos uma *fila clássica* de textos, que contém as sílabas de uma palavra. Se desejássemos criar uma fila vazia,

bastava omitir os valores iniciais e usar somente os colchetes.

- Para *enfileirar* (`enqueue`) elementos em nossa fila, fazemos:

```
f.push("!");
```

Podemos notar que, em JS, assim como em uma pilha, usamos a operação `push` para acrescentar elementos em um fila. Dessa forma, `f.push("!");` enfileira o elemento `!` ao *fim* da fila.

- Para saber qual elemento está na *cabeça* (`head`) de nossa fila, fazemos:

```
f[0];
```

Como a fila usa o objeto *Array*, simplesmente acessamos o elemento no índice zero (`0`) para sabermos a atual *cabeça* da fila. Essa execução retorna `FLA` .

- Para *desenfileirar* (`dequeue`) de nossa fila, fazemos:

```
f.shift();
```

Em `f.shift()` , o primeiro elemento da fila (cabeça) é removido e retornado. Assim, temos como resultado `FLA` . A fila contém agora os elementos `MEN` , `GO` e `!` .

- Para saber a *quantidade de elementos* de nossa fila, fazemos:

```
f.length;
```

Para `f` , a execução desse código resulta em `3` .

- Adicionalmente, podemos usar a operação `indexOf` para *pesquisar elementos* em uma fila também, assim como em uma pilha. O comportamento anteriormente explicado se aplica a essa ED também.

```
f.indexOf("GO");
```

O resultado dessa execução é `1`, pois esse elemento ainda se encontra enfileirado em nossa fila, na segunda posição.

14.4 Lista

Por fim, vamos usar o objeto *Array* como uma *lista*.

- Para *criar* uma lista:

```
var l = [];
```

Esse código cria uma lista vazia chamada `l`.

- Para *adicionar elementos* em nossa lista, fazemos:

```
l.push(2009);
l.push(2019);
l.push(2022);
```

Os códigos anteriores são responsáveis por adicionar elementos do tipo *inteiro* em nossa lista.

- Para *obter um elemento* de nossa lista, fazemos:

```
l[2];
```

Podemos notar que a forma de acesso a um elemento de uma lista em JS é igual à de um vetor, pilha ou fila. Assim, `l[2]` nos retorna o valor `2022`, que é o terceiro e, coincidentemente, o último elemento da lista.

- Para *excluir um elemento* de nossa lista, fazemos:

```
l.splice(2,1);
```

O código anterior usa a função `splice` para realizar a exclusão de elementos. Dessa forma, `splice(2,1);` exclui `2022`. Nessa função, o `2` é a posição de `2022` na *lista* e `1`, a quantidade de elementos a

serem excluídos, então excluímos exatamente `2022`. Essa função ainda possibilita a exclusão de outras maneiras, mas aqui apenas exploraremos essa, por ser a mais simples e, consequentemente a mais utilizada. Para mais detalhes, nas *Referências bibliográficas* temos o link para um conteúdo mais vasto sobre essa função.

- Para saber a *quantidade de elementos* de nossa lista, fazemos:

```
l.length;
```

O resultado dessa execução retorna `2`.

Para finalizar, é válido informar que JS não disponibiliza uma implementação nativa para *lista duplamente encadeada*. Aconselho você a implementá-la como estudo, baseando-se nos códigos em C do capítulo 6, *Lista*.

14.5 Árvore

JS não provê uma implementação nativa de árvore. Devido a isso, vamos ter aqui uma implementação própria de uma árvore. Abordaremos a *árvore binária ordenada*, então, teremos mais uma implementação, já que no capítulo 8, *Árvore*, implementamos os tipos clássica, AVL e N-ária. Como sabemos, JavaScript é uma linguagem Orientada a Objetos (OO), então é bom conhecer um pouco de OO para entender os códigos a seguir. Mais uma vez, aconselho a leitura de *Orientação a Objetos: Aprenda seus conceitos e suas aplicabilidades de forma efetiva* (ISBN: 978-85-5519-213-5).

Vamos definir a estrutura da árvore, assim como suas operações:

```
class No {

    constructor(valor) {
        this._dado = valor;
        this._filhoEsquerdo = null;
```

```
    this._filhoDireito = null;
}

get dado() {
  return this._dado;
}

set dado(valor) {
  this._dado = valor;
}

get filhoEsquerdo() {
  return this._filhoEsquerdo;
}

set filhoEsquerdo(valor) {
  this._filhoEsquerdo = valor;
}

get filhoDireito() {
  return this._filhoDireito;
}

set filhoDireito(valor) {
  this._filhoDireito = valor;
}

insert(valor) {

    if(valor < this.dado) {
        if(this.filhoEsquerdo == null) {
            let novoNo = new No(valor);
            this.filhoEsquerdo = novoNo;
            return;
        } else {
            this.filhoEsquerdo.insert(valor);
            return;
        }
    } else {
```

```
        if(this.filhoDireito == null) {
            let novoNo = new No(valor);
            this.filhoDireito = novoNo;
            return;
        } else {
            this.filhoDireito.insert(valor);
            return;
        }
    }
}

delete(valor) {

    if(valor < this.dado) {
        this.filhoEsquerdo = this.filhoEsquerdo.delete(valor);
        return this;
    } else if(valor > this.dado) {
        this.filhoDireito = this.filhoDireito.delete(valor);
        return this;
    } else {
        if(this.filhoEsquerdo == null && this.filhoDireito == null) {
            return null;
        }

        if(this.filhoEsquerdo == null) {
            return this.filhoDireito;
        } else if(this.filhoDireito == null) {
            return this.filhoEsquerdo;
        }

        var noAuxiliar = this.menorValor(this.filhoDireito);
        this.dado = noAuxiliar.dado;

        this.filhoDireito = this.delete(noAuxiliar.dado);
        return this;
    }
}

menorValor(no) {
```

```
    if(no.filhoEsquerdo === null) {
        return no;
    } else {
        return this.menorValor(no.filhoEsquerdo);
    }
}

search(valor) {

    if (valor == this.dado) {
        console.log(valor + ' pertence à árvore.');
        return;
    } else if (valor < this.dado) {
        if (this.filhoEsquerdo == null) {
            console.log(valor + ' não pertence à árvore.');
            return;
        } else {
            return this.filhoEsquerdo.search(valor);
        }
    } else if (valor > this.dado) {
        if (this.filhoDireito == null) {
            console.log(valor + ' não pertence à árvore.');
            return;
        } else {
            return this.filhoDireito.search(valor);
        }
    }
    console.log(valor + ' não pertence à árvore.');
    return;
}

exibirPreOrdem() {

    console.log(this._dado + " ");

    if (this._filhoEsquerdo != null) {
        this._filhoEsquerdo.exibirPreOrdem();
    }
```

```
        if (this._filhoDireito != null) {
            this._filhoDireito.exibirPreOrdem();
        }
    }

    exibirEmOrdem() {


        if (this._filhoEsquerdo != null) {
            this._filhoEsquerdo.exibirEmOrdem();
        }

        console.log(this._dado + " ");

        if (this._filhoDireito != null) {
            this._filhoDireito.exibirEmOrdem();
        }
    }

    exibirPosOrdem() {


        if (this._filhoEsquerdo != null) {
            this._filhoEsquerdo.exibirPosOrdem();
        }

        if (this._filhoDireito != null) {
            this._filhoDireito.exibirPosOrdem();
        }

        console.log(this._dado + " ");
    }

}
```

O código anterior é responsável por definir as informações que um *nó* possui: `dado`, `filhoEsquerdo` e `filhoDireito`. Também definimos as operações às quais o nó pode ser submetido (e, consequentemente,

a *árvore*): `insert`, `delete` e `search`. Adicionalmente, temos as três formas de navegação, `imprimir_pre_ordem`, `imprimir_em_ordem` e `imprimir_pos_ordem`. Além disso, já explicamos como uma *árvore binária ordenada* se comporta no capítulo 8. Então, revise-o e acompanhe a execução do código disponibilizado.

14.6 Set

JS provê um objeto chamado ***Set*** para podermos criar e manipular essa estrutura.

- Para *criar* um *set*, fazemos:

```
var s = new Set();
```

O código anterior cria um *set* vazio chamado `s`.

- Para *adicionar elementos* ao nosso *set*, fazemos:

```
s.add(1980);
s.add(1981);
s.add(1982);
s.add(1983);
s.add(1981);
s.add(2009);
```

Como essa ED é a implementação de conjuntos (conceito matemático), a segunda tentativa de adicionar o `1981` (`s.add(1981)`) não surte efeito, ou seja, `1981` não é adicionado no *set*, pois esse valor já tinha sido adicionado anteriormente. Nesse exemplo, nosso *set* foi iniciado com números.

- Para saber se um elemento *está contido no set*, fazemos:

```
s.has(1992);
```

O código anterior tem como resultado o valor `false`, pois `1992` não é um valor pertencente ao nosso *set*. Caso passemos um valor pertencente ao *set*, obteremos o valor `true`.

- Para *remover um elemento* de nosso *set*, fazemos:

```
s.delete(1983);
```

A execução desse código remove o elemento `1983`. Caso o elemento passado como parâmetro não exista no *set*, nada acontece.

- Para saber a *quantidade de elementos* de nosso *set*, fazemos:

```
s.size;
```

Inicialmente, após os seis *adds*, o tamanho era `5`, mas como executamos um *remove*, o tamanho atual é `4`.

- Para *iterar* sobre os elementos nessa ED em JS, fazemos:

```
for (var valor of s) {
  console.log(valor);
}
```

Um resultado dessa execução pode ser: `1980`, `1981`, `1982` e `2009`.

14.7 Tabela de dispersão (Hashtable)

O objeto em JS para tabela de dispersão é o *map*.

- Para *criar* um *map*, fazemos:

```
var m = new Map();
```

No código anterior, criamos um *map* chamado `m`, que ainda está vazio, mas o preencheremos a seguir.

- Para *adicionar elementos* ao nosso *map*, fazemos:

```
m.set("brasil",5);
m.set("alemanha",4);
m.set("itália",4);
m.set("argentina",2);
m.set("França",2);
m.set("uruguai",2);
m.set("inglaterra",1);
m.set("espanha",1);
```

Nos códigos, adicionamos valores ao nosso *map*. Ele tem as *chaves* em texto e os *valores* em números.

- Para saber se um elemento *está contido* no nosso *map*, fazemos:

```
m.has("brasil");
```

O código anterior usa o `has` para realizar essa operação. Como resultado dessa execução, obteremos o valor `true`, pois `'brasil'` é uma chave pertencente ao nosso *map*. É válido ressaltar que `has` sempre deve receber uma *chave* como parâmetro.

- Para *obter um valor* do nosso *map*, fazemos:

```
m.get("alemanha");
```

Como resultado dessa execução, obteremos `4`, pois esse é o valor associado à chave `alemanha`. Se a chave passada como parâmetro não estiver contida no *map*, `undefined` é retornado.

- Para *alterar um valor* do nosso *map*, fazemos:

```
m.set("argentina", 3);
```

Como a chave `argentina` já existe no *map*, seu valor é modificado para `3`. Caso não existisse, a inclusão dela seria feita.

- Para *remover* um elemento de nosso *map*, fazemos:

```
m.delete("itália");
```

A execução desse código apaga a chave `itália` e o valor associado a ela. Se a chave passada como parâmetro não pertencer ao *map*, nada acontece.

- Para saber a *quantidade de elementos*, fazemos:

```
m.size;
```

Inicialmente, após os oito *sets*, o tamanho era `8`, mas como executamos um *remove*, o tamanho atual é `7`.

Por fim, temos duas formas de percorrer nosso *map*: pelas chaves e pelos valores. A seguir, vemos essas duas abordagens.

```
//Pelas chaves
for (var key of m.keys()) {
  console.log(key);
}

//Pelos valores
for (var key of m.values()) {
  console.log(key);
}
```

Alguns possíveis resultados dessas execuções seriam, respectivamente: `brasil`, `alemanha`, `argentina`, `França`, `uruguai`, `inglaterra` e `espanha`; e `5`, `4`, `2`, `2`, `2`, `1` e `1`.

Para saber mais...

Caso precise de mais informações sobre estruturas de dados em JS, aconselho a leitura dos conteúdos abaixo. Eles foram a base para este capítulo.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

Parte 3: Conclusão e apêndices

Capítulo 15
Chegamos ao fim

Chegamos ao fim da jornada de entender o que são e como funcionam as estruturas de dados. Essa caminhada foi por vezes árdua, admito, pois alguns conceitos podem ter se demonstrado complexos inicialmente. Além disso, os códigos em `C` podem ter sido mais difíceis de entender em alguns momentos. Mas com prática e amadurecimento no processo de desenvolvimento, tudo se encaixará com mais clareza. No fim, espero que este livro tenha ajudado a responder às seguintes perguntas:

- Qual a ED que vai me ajudar a resolver o meu problema?
- Qual classe de minha linguagem de trabalho devo usar para utilizar a ED escolhida?
- Como posso utilizar a classe escolhida para solucionar meu problema?

Para ajudar a responder à segunda pergunta, as listagens a seguir agrupam as EDs de acordo com as classes que as implementam em cada linguagem.

Java

1. `Stack` : pilha
2. `LinkedList` : fila e lista
3. `ArrayList` : lista
4. `HashSet` , `LinkedHashSet` , `TreeSet` : set
5. `HashMap` , `Hashtable` : tabela de dispersão

C#

1. `Stack` : pilha
2. `Queue` : fila
3. `List` : lista
4. `HashSet` , `SortedSet` : set
5. `Hashtable` , `Dictionary` : tabela de dispersão

Python

1. `list` : pilha, fila, lista
2. `set` : set
3. `dict` : tabela de dispersão

JS

1. `Array` : pilha, fila, lista
2. `Set` : set
3. `Map` : tabela de dispersão

Obter essas respostas ajudará a utilizar as EDs da melhor forma e isso levará a uma melhor solução do problema em questão. Além disso, entender os conceitos básicos, como tipo de dado, tipo abstrato de dado, etc. também ajuda a desenvolver o melhor código e, consequentemente, a melhor solução para o problema.

Por último, deixo uma tabela que sumariza todo o conteúdo que este livro abordou. A ideia é que ela sirva como um guia de consulta:

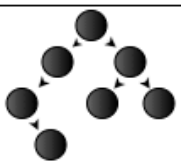
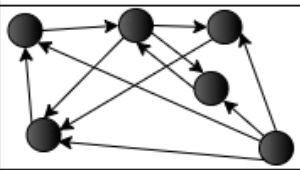
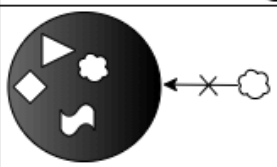
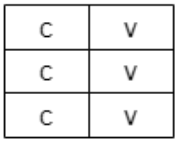
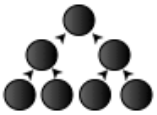
Estrutura de Dado	Ilustração	Características	Aplicação
Vetor		- Tamanho fixo - Acesso sequencial - Unidimensional - Homogêneo	Conjunto de elementos limitado e de tamanho previamente conhecido
Matriz		- Tamanho fixo - Acesso sequencial - Multidimensional - Homogênea	Representação de planos cartesianos
Pilha		- Dinâmica (aconselhável) - LIFO - Homogênea (aconselhável)	Ctrl+z
Fila		- Dinâmica (aconselhável) - FIFO - Homogênea (aconselhável)	Fila de impressão
Lista		- Dinâmica (aconselhável) - Manipulação livre e posicional - Homogênea (aconselhável)	Comentários de uma rede social
Árvore		- Dinâmica - Hierárquica - Homogênea (aconselhável)	Sistema de arquivos em Sistemas Operacionais
Grafo		- Dinâmico - Relacional - Homogêneo (aconselhável)	Relacionamentos em redes sociais
Conjunto (Set)		- Dinâmico - Elementos únicos - Homogêneo	Conjunto de números de identificação de pessoas (CPF)
Tabela de Dispersão (HashTable)		- Dinâmico - Chave/Valor - Homogêneo (aconselhável)	Nos compiladores, mapear as palavras reservadas de linguagens de programação para suas ações.
Heap		- Dinâmico - Hierárquico - Ordenado - Homogêneo (aconselhável)	Escalonamento de tarefas prioritárias

Figura 15.1: Resumo

Capítulo 16
Referências bibliográficas

16.1 Livros

ASCENCIO, Ana F. G.; ARAÚJO, Graziela S. *Estrutura de dados*: algoritmos, análise de complexidade e implementações em Java e C/C++. São Paulo: Pearson Universidades, 2010.

ASCENCIO, Ana F. G.; CAMPOS, Edilene A. V. *Fundamentos da programação de computadores*: Algoritmos, Pascal, C/C++ (padrão ANSI) e Java. 3. ed. São Paulo: Pearson Universidades, 2012.

PIVA JUNIOR, Dilermando; NAKAMITI, Gilberto S.; BIANCHI, Francisco; FREITAS, Ricardo L.; XASTRE, Leandro A. *Estrutura de Dados e Técnicas de Programação*. 1. ed. Rio de Janeiro: Elsevier, 2014.

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José L. *Introdução a estrutura de dados* - com técnicas de programação em C. 4. ed. Rio de Janeiro: Elsevier/Campus, 2004.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algoritmos*:

Teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

EDELWEISS, Nina; GALANTE, Renata. *Estruturas de dados*. Livros Didáticos Informática UFRGS, v. 18. 1. ed. Porto Alegre: Bookman, 2008.

GOODRICH, Michael T.; TAMASSIA, Roberto. *Estruturas de dados e algoritmos em Java*. 4. ed. Porto Alegre: Bookman, 2007.

GRONER, Loiane. *Learning JavaScript Data Structures and Algorithms*. 3. ed. Birmingham: Packt Publishing, 2018.

KNUTH, Donald E. *The Art of Computer Programming*: Volume 1 - Fundamental Algorithms. 3. ed. Addison-Wesley, 1997.

MAIDA, João Paulo. *Teoria dos Grafos*: Uma abordagem prática em Java. São Paulo: Casa do Código, 2020.

PUGA, Sandra; RISSETTI, Gerson. *Lógica de programação e Estruturas de Dados com Aplicações em Java*. 3. ed. São Paulo: Pearson Universidades, 2016.

SZWARCFITER, Jayme L.; MARKENZON, Lilian. *Estruturas de dados e seus algoritmos*. 3. ed. Rio de Janeiro: LTC, 2010.

SCHILDT, Herbert. *C*: completo e total. 3. ed. São Paulo: Pearson Makron Books, 1997.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Pearson/Makron Books, 1995.

ZIVIANI, Nivio. *Projeto de algoritmos com implementações em Pascal e C*. 4. ed. São Paulo: Pioneira Informática, 1999.

16.2 Links

Java

JAVA DOCUMENTATION. *Arrays*. Disponível em: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html. Acesso em 15 nov. 2022.

JAVA DOCUMENTATION. *ArrayList*. Disponível em: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. Acesso em 16 nov. 2022.

JAVA DOCUMENTATION. *HashMap*. Disponível em: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. Acesso em 18 nov. 2022.

JAVA DOCUMENTATION. *HashSet*. Disponível em: https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html. Acesso em 17 nov. 2022.

JAVA DOCUMENTATION. *LinkedList*. Disponível em: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html. Acesso em 16 nov. 2022.

JAVA DOCUMENTATION. *Stack*. Disponível em: https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html. Acesso em: 15 nov. 2022.

CSharp

MICROSOFT DOCUMENTATION. *Arrays*. Disponível em: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/. Acesso em 19 nov. 2022.

MICROSOFT DOCUMENTATION. *Dictionary*. Disponível em: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-7.0. Acesso em 21 nov. 2022.

MICROSOFT DOCUMENTATION. *HashSet*. Disponível em: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-7.0. Acesso em 21 nov. 2022.

MICROSOFT DOCUMENTATION. *List*. Disponível em: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-7.0. Acesso em 19 nov. 2022.

MICROSOFT DOCUMENTATION. *Multidimensional Arrays*. Disponível em: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/multidimensional-arrays. Acesso em 19 nov. 2022.

MICROSOFT DOCUMENTATION. *Queue*. Disponível em: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1?view=net-7.0. Acesso em 19 nov. 2022.

MICROSOFT DOCUMENTATION. *Stack*. Disponível em: https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1?view=net-7.0. Acesso em 19 nov. 2022.

Python

PYTHON DOCUMENTATION. *Array*: efficient arrays of numeric values. Disponível em: https://docs.python.org/3.7/library/array.html#module-array. Acesso em 23 nov. 2022.

KUMAR, Akash. *Data Structures You Need To Learn In Python*. Edureka, 9 abr. 2022. Disponível em: https://www.edureka.co/blog/data-structures-in-python. Acesso em 22 nov. 2022.

PYTHON DOCUMENTATION. *Deque*. Disponível em: https://docs.python.org/3/library/collections.html#collections.deque. Acesso em 29 nov. 2022.

PYTHON DOCUMENTATION. *Dict*. Disponível em: https://docs.python.org/3/library/stdtypes.html#typesmapping. Acesso em 1 nov. 2022.

PYTHON DOCUMENTATION. *List*. Disponível em: https://docs.python.org/3/tutorial/datastructures.html#more-on-lists. Acesso em 1 dez. 2022.

PYTHON DOCUMENTATION. *Queue*. Disponível em: https://docs.python.org/3/tutorial/datastructures.html?highlight=stack#using-lists-as-queues. Acesso em 29 nov. 2022.

PYTHON DOCUMENTATION. *Set*. Disponível em: https://docs.python.org/3/library/stdtypes.html#set. Acesso em 1 dez. 2022.

PYTHON DOCUMENTATION. *Stack*. Disponível em: https://docs.python.org/3/tutorial/datastructures.html?highlight=stack#using-lists-as-stacks. Acesso em 23 nov. 2022.

PYTHON DOCUMENTATION. *Tuple*. Disponível em: https://docs.python.org/3/library/stdtypes.html#tuple. Acesso em 2 dez. 2022.

JS

MOZILLA DEVELOPER NETWORK. *Array*. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array. Acesso em 19 dez. 2022.

MOZILLA DEVELOPER NETWORK. *Map*. Disponível em: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map. Acesso em 20 dez. 2022.

MOZILLA DEVELOPER NETWORK. *Set*. Disponível em: https://developer.mozilla.org/en-

US/docs/Web/JavaScript/Reference/Global_Objects/Set. Acesso em 21 dez. 2022.

Capítulo 17
Apêndice I: Ponteiro em C

Como explicado no capítulo 2, *Conceitos básicos*, um **ponteiro** é um tipo de variável onde se armazenam endereços de memória, os quais podem apontar para outras variáveis. Neste apêndice, vamos explorar como criar e manipular ponteiros em C, pois este conceito é bastante utilizado no livro.

Para criar e manipular ponteiros em C, dois operadores devem ser utilizados: `*` e `&`, respectivamente. O trecho de código a seguir ilustra como utilizar esses operadores.

```
/*
 * ExemploPonteiro.c
 *
 * Author: Thiago Leite
 */
#include<stdio.h>

int main() {

    int i;

    int *ponteiro_int;

    i = 10;

    ponteiro_int = &i;

    printf("ponteiro_int: %p", ponteiro_int);

    printf("\n");

    printf("ponteiro_int: %d", *ponteiro_int);
```

```
    return 0;
}
```

No código anterior, usamos o operador `*` para definir um ponteiro em `*ponteiro_int` — no caso um ponteiro para inteiros. Com isso, estamos dizendo: "crie uma variável onde poderei, por exemplo, armazenar o endereço de memória de um inteiro". Assim, quando a linha `ponteiro_int = &i;` é executada, ela usa o operador `&` para obter o endereço da variável `i`, que é um inteiro (`int`).

Devido a isso, a linha `printf("ponteiro_int: %p", ponteiro_int);` não exibe o valor `10` que foi atribuído a `i` na linha `i = 10;`, mas, sim, o endereço de memória onde o `10` está armazenado. Todavia, se precisarmos acessar o valor da variável, e não seu endereço, devemos novamente utilizar o operador `*`, como na linha `printf("ponteiro_int: %d", *ponteiro_int);`. O resultado da execução do código anterior encontra-se na imagem a seguir.

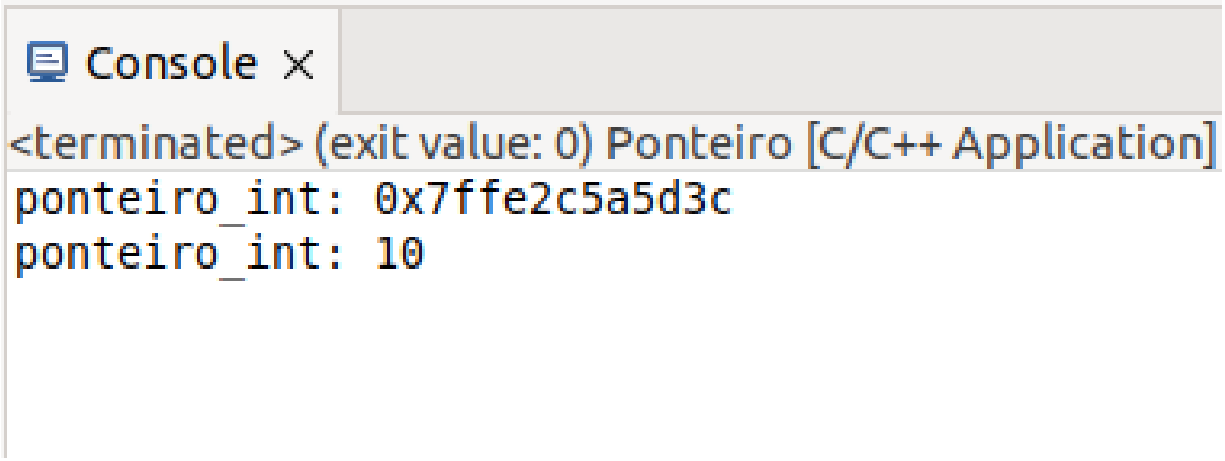


Figura 17.1: Resultado da execução do exemplo de um ponteiro.

É válido ressaltar que, assim como criamos um ponteiro para um `int`, podemos criar um ponteiro para qualquer outro tipo primitivo, basta seguir o mesmo princípio que utilizamos neste apêndice. Além disso, podemos criar ponteiros para funções (`function`), registros (`struct`) e até mesmo ponteiros de ponteiros. Entretanto, essa última possibilidade não é necessária para este livro e não é

abordada, mas, caso deseje, aconselho a leitura de livros específicos da linguagem C. Já ponteiros para *struct* serão abordados no Apêndice II. O exemplo que foi abordado neste apêndice é suficiente para este livro.

Capítulo 18
Apêndice II: Struct em C

A linguagem C disponibiliza a ***struct*** com a finalidade de criar uma estrutura que tenha a capacidade de aglutinar variáveis que, juntas, terminaram por representar com conceito mais amplo. Neste caso, podemos dizer que é um conceito "complexo" e "heterogêneo", pois pode possuir várias variáveis de tipos diferentes. É de se esperar que estas tenham uma relação representacional entre si.

Para definir uma *struct*, devemos utilizar a palavra reservada `struct`. O código a seguir exemplifica melhor.

```
struct pessoa {
  char nome[100];
  int idade;
};
```

No código anterior, foi definido uma `struct` chamada `pessoa`. Esta possui dois membros (variáveis), no caso `nome`, que é um *array* de caracteres, e `idade`, que é um inteiro.

Para criar uma variável do tipo desta `struct` podemos fazer a seguinte declaração:

```
struct pessoa Pessoa;
```

Somente após isso podemos de fato começar a usar nossa *struct* `Pessoa`, através da variável `Pessoa`. Para isso, vamos atribuir alguns valores:

```
Pessoa.nome = "Yoda";
Pessoa.idade = 900;
```

No código, percebe-se que, para acessar os membros de uma *struct*, usa-se o `.`; já as atribuições são da forma que já somos acostumados a fazer com variáveis simples.

Embora essa seja a forma mais comum de se definir e usar *structs*, podemos também criar ponteiros para *structs*. Neste caso, fazemos da forma que já sabemos: usar o operador `*`.

```
struct pessoa *Pessoa;
```

O código anterior agora não cria uma *struct* de forma simples, mas sim um *ponteiro* que aponta para a *struct Pessoa* e o disponibiliza através da variável `Pessoa`. Devido a isso, o operador `&` pode também agora ser utilizado, caso necessário. Além disso, a criação desse ponteiro leva a uma outra mudança: não mais utilizamos o `.` para acessar os membros da *struct*, mas, sim, o operador `->`. Veja a seguir:

```
Pessoa->nome = "Yoda";
Pessoa->idade = 900;
```

Para finalizar, podemos usar a palavra reservada `typedef` que C disponibiliza para tornarmos a maneira de criar variáveis do tipo da *struct* mais simples. Usando como exemplo nossa *struct* `pessoa`, podemos mudar sua definição para uma das opções a seguir:

```
typedef struct_pessoa {
  char nome[100];
  int idade;
} Pessoa;

struct pessoa {
  char nome[100];
  int idade;
};
typedef struct pessoa Pessoa;
```

Após escolher alguma das possibilidades anteriores, agora é só fazer como no código a seguir para criar variáveis do tipo desta *struct*:

```
Pessoa pessoa;
```

ou

```
Pessoa *pessoa;
```

Capítulo 19
Apêndice III: Funções nativas

Algumas funções que C disponibiliza de forma nativa são muito úteis no contexto de nosso livro. Este apêndice lista e explica as funções utilizadas neste livro.

sizeof

Embora na verdade `sizeof` seja um operador e não uma função, ele se comporta de maneira muito similar a uma e, por isso, ele está neste apêndice. Basicamente ele nos fornece em tempo de compilação o tamanho em bytes de variáveis ou tipos de dados. A seguir, os exemplos elucidarão seu uso e comportamento.

```
int i;

printf("%d", sizeof i);
printf("%f", sizeof (float));
```

O resultado da execução desse código é, respectivamente, `2` e `8`, que são os bytes que cada tipo de dados ocupa na memória. No caso, o operador `sizeof` foi utilizado com a variável `i` e diretamente com o tipo de dado `float`.

Também é possivel utilizar este operador com tipos de dados complexos, como a *struct*. A seguir, temos um exemplo:

```
struct numeros {
  int i;
  float f;
};
typedef struct numeros Numeros;

printf("%f", sizeof(Numeros));
```

O resultado da execução do código anterior é `10`, pois, separadamente, `int` e `float` têm, respectivamente, `2` e `8` bytes. Como usamos `sizeof` na *struct* `Numeros`, tal valor foi obtido.

Para finalizar, podemos dizer que o uso dos `()` é opcional, mas aconselhado.

malloc

A função `malloc()` pode ser utilizada quando for necessário trabalhar com a alocação dinâmica de memória. Ela recebe como parâmetro a quantidade de bytes que deve ser alocada. Dessa forma, ao executar, ela retorna um ponteiro para o primeiro byte de memória livre correspondente ao tamanho solicitado. Geralmente, o uso de `malloc` é em conjunto com o operador `sizeof`. A seguir, temos exemplos.

```
int *i;

i = malloc(sizeof(int));
```

No código anterior, é alocado um espaço de `2` bytes para um inteiro, o qual é referenciado através de um *ponteiro*.

```
struct numeros {
  int i;
  float f;
};
typedef struct numeros Numeros;

Numeros *numeros;

numeros = malloc(sizeof(Numeros));
```

No código anterior, é alocado um espaço de `10` bytes para um *struct* do tipo *Numeros*, o qual é referenciado através de um ponteiro. São alocados `10` bytes porque `int` e `float` têm, respectivamente, `2` e `8` bytes. E no caso, a *struct* possui estes tipos de dados como membros.

free

A função `free()` é responsável por liberar um determinado espaço de memória que foi alocado dinamicamente. Ou seja, é de se esperar que para cada `malloc()` exista um `free()` correspondente. Esta função recebe como parâmetro um *ponteiro* que aponta para o espaço de memória que foi alocado.

Levando em consideração os exemplos de alocação da seção anterior, teríamos:

```
free(i);

free(numeros);
```

Com esse código, os respectivos ponteiros `i` e `numeros` tiveram seus espaços de memória desalocados. Note que foi passado o *ponteiro* em si, não seu endereço de memória, no caso `&i` ou `&numeros`. Caso isso fosse feito, um erro ocorreria, pois a função espera receber como parâmetro o ponteiro como um todo e não seu endereço.

Capítulo 20
Apêndice IV: Recursividade

Na programação, o termo **recursividade** é usado para descrever o comportamento com o qual uma função (programação estruturada) ou método (programação orientada a objetos) executa, repetidamente, uma chamada a si mesmo.

Uma execução recursiva é composta por três componentes:

- Um teste de parada;
- Uma operação a ser realizada;
- A chamada recursiva para uma versão simplificada do problema original.

Uma observação relevante sobre o uso de recursão é que o teste de parada sempre tem que estar antes da chamada recursiva, senão o programa entra em loop. A partir da lista anterior, podemos chegar ao seguinte pseudocódigo:

```
função/método(...) {

  if(condição) { /*Teste de parada*/
    código /*Operação a ser realizada*/
    função/método(...); /*Chamada recursiva*/
  }
}
```

Para entender melhor como usar a *recursividade*, vamos ver um exemplo básico: um `for` de forma recursiva. Sabemos que o `for` é uma estrutura de iteração clássica e pode ser usado da seguinte maneira em `C`:

```
void for_function() {

 int i;
 for (i = 1; i <= 10; i++) {
```

```
   printf("%d ", i);
 }
}
```

O código anterior imprime os números de `1` até `10`. Sua versão recursiva, que imprime exatamente o mesmo resultado, seria:

```
void for_recursive(int n) {

  if (n <= 10){

   printf("%d ", n);
   for_recursive(n+1);
  }
}
```

Assim, o código completo que usa essa versão recursiva da função `for_function` chamada de `for_recursive` é:

```
#include <stdio.h>

void for_recursive();

int main()
{
    for_recursive(1);

    return 0;
}

void for_recursive(int n) {

  if (n <= 10){

   printf("%d ", n);
   for_recursive(n+1);
  }
}
```

Exemplos clássicos de uso de *recursividade* são os cálculos de Fibonaccis, fatoriais e a criação de fractais.

Por fim, embora inicialmente possa parecer mais complexo pensar de forma recursiva, com o passar do tempo isso se torna mais fácil, pois chamadas recursivas têm a capacidade de simplificar lógicas mais complexas. Entretanto, a recursividade pode causar um maior consumo de memória, devido ao aumento da pilha de execução (ver capítulo 2, seção *Memória*).